PowerShell Basics

Windows Command Shell for Automation & Scripting

Windows	Command Shell	Automation	Scripting	.NET	Object-Based	Terminal
Built for Windows	Powerful CLI	Save time & work	Write powerful scripts	Built on .NET Framework	Works with objects	Modern & Powerful

PowerShell Terminal

```
PS C:\Users\CodeSardar> Get-Host

Name                : ConsoleHost
Version             : 5.1.22621.3672
InstanceId          : 2aBb3f2b-7c71-4b3c-9d78-7e6d8e2f6f1a
UI                  : System.Management.Automation.Internal.Host.InternalHostUserInter-
                      terface
CurrentCulture      : en-IN
CurrentUICalture    : en-IN
PrivateData         : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
DebuggerEnabled     : True
IsRunspacePushed    : False

PS C:\Users\CodeSardar>
```

1. BASIC COMMAND SYNTAX

PowerShell commands follow the Verb-Noun pattern.

Verb-Noun -Parameter Value

- **Verb** - Action to perform (Get, Set, New, Remove, Start, Stop)
- **Noun** - The object to act on (Process, Service, Item, Content)
- **Paramater** - Additional options to customize the command
- **Value** - The value for the parameter

Examples:

```
Get-Process                                         # Get all running processes
Get-Service -Name "Spooler"                         # Get a specific service
Get-ChildItem -Path "C:\" -Recurse                  # List items in a folder
New-Item -Path "C:\Test" -ItemType Directory        # Create a new folder
```

2. COMMON CMDLETS (VERB-NOUN)

Verb	Meaning	Examples
Get	Retrieve data	Get-Process, Get-Service
Set	Modify data	Set-Location, Set-Service
New	Create new	New-Item, New-Service
Remove	Delete / Remove	Remove-Item, Remove-Process
Start	Start something	Start-Service, Start-Process
Stop	Stop something	Stop-Service, Stop-Process
Clear	Clear data	Clear-Host, Clear-Item
Export	Export data	Export-Csv, Export-Clixml
Import	Import data	Import-Csv, Import-Clixml

WHAT IS POWERSHELL?

PowerShell is a powerful command-line shell and scripting language from Microsoft. It is built on the .NET framework and is designed for system administration, automation, and configuration management.

KEY FEATURES

- **Object-Based:** Works with .NET objects, not just text
- **Pipeline:** Pass objects from one command to another
- **Scripting:** Write scripts to automate complex tasks
- **Remote Management:** Manage remote systems easily
- **Extensible:** Add new commands and modules
- **Cross-Platform:** PowerShell 7+ runs on Windows, Linux, macOS

3. PIPELINE & OUTPUT

PowerShell uses pipeline (|) to send objects from one command to another.

Example:

```
Get-Process | Where-Object {$_.CPU -gt 100} |
Sort-Object CPU -Descending | Select-Object
Name, CPU
```

Output is formatted as tables, lists, or custom views.

4. VARIABLES

Use $ to create variables.

Examples:

```
$name    = "CodeSardar"
$age     = 25
Write-Host "Name: $name, Age: $age"
Get-Process -Name $name
```

5. COMMENTS

Use # for single-line comments
<# #> for multi-line comments

Examples:

```
# This is a single-line comment
Write-Host "Hello PowerShell"
<#
This is
a multi-line
comment
#>
```

WHEN TO USE POWERSHELL?

- System Administration & Management
- Automate repetitive tasks
- Manage Windows Services, Processes, Files, Registry
- Work with Active Directory, Azure, and more

6. COMMON DATA TYPES

Type	Example
String	"Hello World"
Int32	123
Boolean	$true / $false
Array	1,2,3
HashTable	@(Name="John"; Age=25)
Object	Get-Process returns objects

7. GET STARTED

1. Open PowerShell (Search: PowerShell)
2. Try simple commands: Get-Process, Get-Service
3. Use Get-Help command for help
4. Explore and automate!

Get-Help Examples

```
Get-Help Get-Process
Get-Help Get-Service -Examples
Get-Help Get-Process -Full
Get-Help about_Pipelines
```

PRO TIP

Use Tab key for auto-completion and faster typing.

Tab

NOTE

PowerShell is case-insensitive and remembers your command history. Use ↑ ↓ keys to navigate history.

SHORTCUTS

Ctrl + C	Cancel command
Ctrl + L	Clear screen
Ctrl + R	Search history
F7	Show history

PowerShell Setup

Install, Open & Get Started with **PowerShell**

What is PowerShell?

PowerShell is a powerful command-line shell and scripting language from Microsoft. It helps you automate tasks, manage systems and work more efficiently.

1 Windows PowerShell

Built-in with Windows (Windows 7 and later)

`powershell.exe`

2 PowerShell 7 (Core)

Latest cross-platform version (Windows, Linux, macOS)

`pwsh`

3 Terminal

Modern terminal to run PowerShell (Windows Terminal)

`wt.exe`

4 ISE (Integrated Scripting Environment)

GUI-based editor for writing and testing scripts

`powershell_ise.exe`

5 VS Code

Best code editor for writing PowerShell scripts

`code .`

Install PowerShell 7

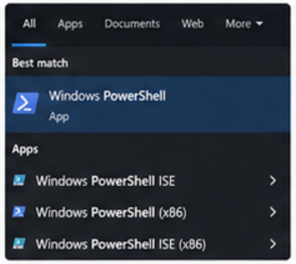
Recommended

1. Go to: https://aka.ms/powershell
2. Download the MSI installer (Windows x64)
3. Run the installer and follow the steps
4. After installation, open Terminal or PowerShell and type: `pwsh --version`

How to Open PowerShell

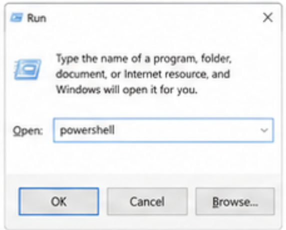
Start Menu Search

Search "PowerShell" in Start Menu and open.



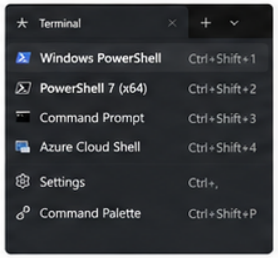
Run Dialog

Press Win + R, type powershell or pwsh and press Enter.



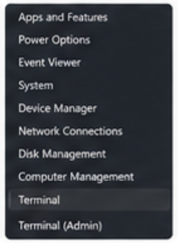
Terminal

Open Windows Terminal and select PowerShell or pwsh.



Right Click Menu

Right click on Start button and choose Terminal or Terminal (Admin).



Check Installed Versions

powershell.exe

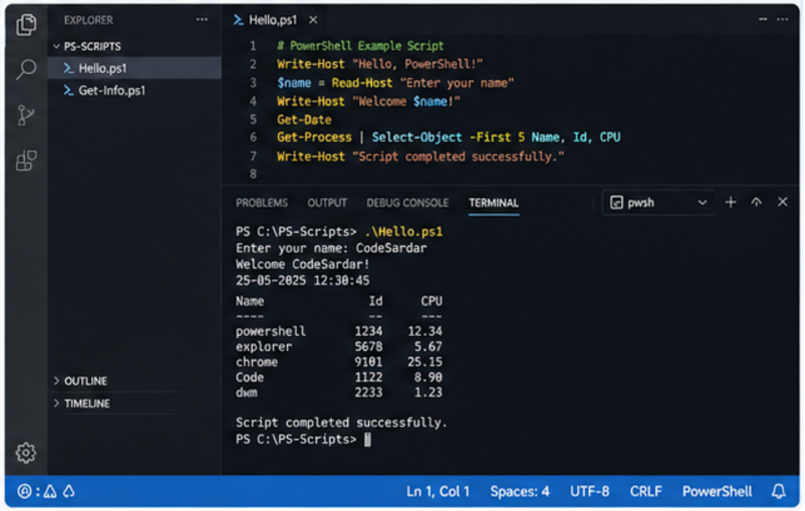
```
PS C:\> $PSVersionTable.PSVersion
Major  Minor  Patch  PreReleaseLabel
-----  -----  -----  ---------------
5      1      22621
```

pwsh (PowerShell 7)

```
PS C:\> $PSVersionTable.PSVersion
Major  Minor  Patch  PreReleaseLabel
-----  -----  -----  ---------------
7      4      1
```

If PowerShell 7 is installed, use `pwsh` to open it.

PowerShell Scripting in VS Code



Common PowerShell Commands (Quick Start)

Command	Description
Get-Help	# Get help about cmdlets
Get-Command	# List all commands
Get-Process	# List running processes
Get-Service	# List services
Get-Date	# Show current date & time
Get-ChildItem	# List files & folders
Set-Location	# Change directory
Get-Content	# Read file content
Set-Content	# Write to file
Get-ExecutionPolicy	# Check execution policy
New-Item	# Create new item
Remove-Item	# Delete item
Copy-Item	# Copy item
Move-Item	# Move/Rename item
Invoke-Command	# Run command remotely

Pro Tip Use `Tab` key for auto-completion, ↑/↓ for command history and `Get-Help <command>` for help.

Get-Help

PowerShell Help System

Get-Help

Displays help about PowerShell cmdlets, functions, scripts, and concepts.
Your built-in guide to PowerShell documentation!

Quick Syntax

```
Get-Help <command-name> [-Examples] [-Detailed] [-Full] [-Online]
```

Basic Usage

Get help for any PowerShell command.

```
Get-Help Get-Process
```

- Discover parameters
- Understand usage
- See examples
- Get detailed info

What is Get-Help?

Get-Help is your built-in help system that provides detailed information about PowerShell cmdlets, functions, and topics.

Example: Get-Help Get-Process

Windows PowerShell

```
PS C:\> Get-Help Get-Process

NAME
    Get-Process

SYNOPSIS
    Gets the processes that are running on the local computer or a remote computer.

SYNTAX
    Get-Process [[-Name] <String[]>] [-ComputerName <String[]>] [-FileVersionInfo] [-Module]
      [-IncludeUserName] [-InputObject <Process[]>] [<CommonParameters>]

DESCRIPTION
    The Get-Process cmdlet gets the processes that are running on the local computer or a remote computer.
    Without parameters, this cmdlet gets all processes on the local computer.

 -- More -->
```

Get-Help Parameters

-Examples

Shows examples of how to use the command.

```
PS C:\> Get-Help Get-Process -Examples
EXAMPLES
  PS C:\> Get-Process
  Gets all processes on the local computer.
```

-Detailed

Shows detailed information about parameters and usage.

```
PS C:\> Get-Help Get-Process -Detailed
PARAMETERS
  -Name <String[]>
    Specifies the process names.
```

-Full

Shows complete help including parameter types, notes, and more.

```
PS C:\> Get-Help Get-Process -Full
NOTES
  Additional information about
  the command...
```

-Online

Opens online help in your default browser.

```
PS C:\> Get-Help Get-Process -Online

Opening online help for
Get-Process...
```

1 Examples

Show usage examples

```
Get-Help Get-Process -Examples
```

```
PS C:\> Get-Help Get-Process -Examples
EXAMPLES
 Example 1: Get all processes
 PS C:\> Get-Process

 Example 2: Get process by name
 PS C:\> Get-Process -Name notepad

 Example 3: Get process on remote computer
 PS C:\> Get-Process -ComputerName Server01
```

2 Detailed

Show detailed information

```
Get-Help Get-Process -Detailed
```

```
PS C:\> Get-Help Get-Process -Detailed
PARAMETERS
  -Name <String[]>
    Specifies the process names.

  -ComputerName <String[]>
    Specifies the remote computers.

  -FileVersionInfo [<SwitchParameter>]
    Includes file version info.
  ...
```

3 Full

Show complete help

```
Get-Help Get-Process -Full
```

```
PS C:\> Get-Help Get-Process -Full
NAME
  Get-Process
SYNOPSIS
  Gets the processes that are running...
SYNTAX
  Get-Process [[-Name] <String[]>] ...
PARAMETERS
 -Name <String[]>
   Specifies the process names.
 ...
```

4 Online

Open online documentation

```
Get-Help Get-Process -Online
```

```
PS C:\> Get-Help Get-Process -Online

Opening online help for
Get-Process...
```

Opening in your default browser

Microsoft Learn
Get-Process
(Microsoft.PowerShell.Management)
Gets the processes that are running on the local computer or a remote computer.

Documentation & Resources

Built-in Help
Always available offline
Fast and integrated

Microsoft Learn
Official documentation
Detailed articles & guides

PowerShell Gallery
Community resources
Modules & scripts

Quick Tips

- Use Tab key after Get-Help <command> to auto-complete.
- Use Get-Help about_* for help topics (e.g., about_Functions).
- Use Update-Help to update help content (requires internet).
- Help is context-aware and always up-to-date.

Common Help Topics

about_Parameters	about_Providers	about_Modules
about_Functions	about_Workflows	about_Scripts
about_Aliases	about_Signing	about_Remote

Help vs Other Methods

- Get-Help = Built-in PowerShell help
- man / --help = External command help
- Read-Host = Get user input
- Invoke-Expression = Run dynamic commands

Pro Tip

Get-Help is your best friend in PowerShell!
Explore, learn, and become a PowerShell pro.

Get-Help is just a command away! 🚀

Get-Command

— Discover and Get All Commands in PowerShell —

What is Get-Command?

Get-Command gets all commands in PowerShell.
It helps you find cmdlets, functions, aliases, scripts and external commands.

Command Types

- **Cmdlet** – Built-in commands
- **Function** – User defined
- **Alias** – Shortcut commands
- **Script** – .ps1 files
- **Application** – External programs

Basic Syntax

```
Get-Command [parameters]
```

-Name	Specify command name (wildcards allowed)
-Verb	Filter by verb (e.g. Get, Set, New)
-Noun	Filter by noun (e.g. Process, Service)
-Module	Filter by module name
-CommandType	Filter by command type
-All	Show all commands (including duplicates)

Example: Get All Commands

Windows PowerShell

```
PS C:\> Get-Command

CommandType    Name                       Version    Source
-----------    ----                       -------    ------
Cmdlet         Add-AppxPackage            1.0.0.0    Appx
Cmdlet         Add-AppxProvisionedPackage 1.0.0.0    Dism
Function       Add-Content                7.4.1.0    Microsoft.PowerShell.Management
Cmdlet         Add-Job                    7.4.1.0    Microsoft.PowerShell.Core
Alias          ac                                    -> Add-Content
Cmdlet         Add-Member                 7.4.1.0    Microsoft.PowerShell.Utility
...
```

Shows all commands available in your PowerShell session.

Quick Examples

`Get-Command`	Get all commands
`Get-Command -Name Get-Process`	Find command by name
`Get-Command -Verb Get`	Get all commands with verb 'Get'
`Get-Command -Noun Process`	Get all commands with noun 'Process'
`Get-Command -Module NetTCPIP`	Get commands from a module
`Get-Command -CommandType Function`	Get all functions
`Get-Command -Name *service*`	Search with wildcard
`Get-Command -All`	Get all commands (including duplicates)

Find Specific Commands

-Name	Find by command name	`Get-Command -Name Get-Process`
-Verb	Find by verb	`Get-Command -Verb Get`
-Noun	Find by noun	`Get-Command -Noun Service`
-Module	Find in a specific module	`Get-Command -Module Microsoft.PowerShell.Management`
-CommandType	Find by type	`Get-Command -CommandType Function`

Example: By Verb and Noun

```
PS C:\> Get-Command -Verb Get -Noun Process

CommandType  Name         Version  Source
-----------  ----         -------  ------
Cmdlet       Get-Process  7.4.1.0  Microsoft.PowerShell.Core
Alias        gps                   -> Get-Process

...
```

Finds all commands with Verb 'Get' and Noun 'Process'.

Example: By Module

```
PS C:\> Get-Command -Module Microsoft.PowerShell.Management

CommandType  Name           Version  Source
-----------  ----           -------  ------
Cmdlet       Get-ChildItem  7.4.1.0  Microsoft.PowerShell.Management
Cmdlet       Get-Content    7.4.1.0  Microsoft.PowerShell.Management
Cmdlet       Get-Item       7.4.1.0  Microsoft.PowerShell.Management
...
```

Shows commands from the specified module.

Command Type Filter

Type	Description	Example
Cmdlet	Built-in PowerShell cmdlet	Get-Process
Function	User defined function	Get-MyData
Alias	Shortcut for a command	gci (alias of Get-ChildItem)
Script	PowerShell script file	MyScript.ps1
Application	External program / executable	notepad.exe

Examples in Action

Command	Purpose
`Get-Command Get-*`	All commands starting with Get-
`Get-Command *Service*`	All commands containing 'Service'
`Get-Command -Verb New`	All commands with verb New
`Get-Command -Noun Item`	All commands with noun Item
`Get-Command -Module *`	All commands from all modules
`Get-Command -CommandType Alias`	All alias commands

Example: Command Types

```
PS C:\> Get-Command -CommandType *

CommandType  Name          Version     Source
-----------  ----          -------     ------
Cmdlet       Get-Date      7.4.1.0     Microsoft.PowerShell.Core
Function     Get-MyReport  1.0         MyModule
Alias        gci                       -> Get-ChildItem
Script       Backup.ps1    1.0         C:\Scripts
Application  notepad.exe   10.0.22631  C:\Windows\System32
...
```

Shows different types of commands.

Example: Get Help for a Command

```
PS C:\> Get-Help Get-Process -Full

NAME
  Get-Process

SYNOPSIS
  Gets the processes that are running on the local computer.

SYNTAX
  Get-Process [[-Name] <String[]>] [-Id <Int32[]>] [options]
...
```

Use Get-Help <command> for more details.

Pro Tips

- Use Tab key for auto-completion.
- Use wildcards (*) to search faster.
- Combine -Verb and -Noun for precise results.
- Use Get-Help <command> for detailed help.
- Use -Module to explore commands in a module.

Remember

Get-Command is your best friend to explore and understand PowerShell commands!

Get-Command = Discover • Explore • Understand Your PowerShell Commands

Tip: Run Get-Command and start exploring!

Get-Process

View Running Processes in PowerShell

What is Get-Process?

Get-Process gets the processes that are running on your local computer.
It shows details like CPU usage, memory usage, PID, process name and start time.

Key Information You Get

- CPU — CPU Usage
- Memory — Memory Usage
- PID — Process ID
- StartTime — Start Time
- Process — Process Info

Basic Syntax

```
Get-Process [parameters]
```

-Name	Filter by process name
-Id	Filter by process ID (PID)
-ComputerName	Remote computer
-FileVersionInfo	Include file version info
-Module	Include associated modules

Quick Overview

- Displays running processes
- Filter by name or ID
- Shows CPU usage and Memory usage
- Provides PID and Start Time
- Works locally and remotely

Example: Get All Processes

Windows PowerShell

```
PS C:\> Get-Process

 Handles   NPM(K)    PM(K)    WS(K)   CPU(s)     Id   SI   ProcessName
 -------   ------    -----    -----   ------     --   --   -----------
 123           12     4520    15600     0.03    776    1   System Idle Process
 310           20     8920    26432     0.19    512    1   System
 456           35    15640    34220     1.25   1824    1   svchost
 580           28    12480    28160     0.25   2040    1   explorer
 421           22     9860    21560     0.12   3052    1   chrome
 215           15     4520    10240     0.05   4120    1   notepad
 ...
```

Displays all running processes with CPU, Memory, PID and more.

Common Usage Examples

`Get-Process`	Get all running processes
`Get-Process -Name chrome`	Get process by name
`Get-Process -Id 3052`	Get process by PID (ID)
`Get-Process -Name notepad*`	Wildcard search by name
`Get-Process -Name svchost | Sort-Object CPU -Descending`	Sort by CPU usage
`Get-Process | Sort-Object PM -Descending | Select-Object -First 5`	Top 5 processes by memory
`Get-Process -ComputerName PC01`	Get processes on remote computer

Filter by Name

```
PS C:\> Get-Process -Name chrome

Handles  NPM(K)    PM(K)    WS(K)  CPU(s)    Id  ProcessName
-------  ------    -----    -----  ------    --  -----------
398          28   120560   152200    1.45  3052  chrome
276          24    95400   128540    0.98  7896  chrome
265          23    91200   121300    0.75  8120  chrome
```

Shows processes with the name 'chrome'.

Filter by ID (PID)

```
PS C:\> Get-Process -Id 3052

Handles  NPM(K)    PM(K)    WS(K)  CPU(s)    Id  ProcessName
-------  ------    -----    -----  ------    --  -----------
398          28   120560   152200    1.45  3052  chrome
```

Shows process with PID 3052.

Top Processes by CPU

```
PS C:\> Get-Process | Sort-Object CPU -Descending | Select-Object -First 5
Handles  NPM(K)    PM(K)    WS(K)  CPU(s)    Id  ProcessName
-------  ------    -----    -----  ------    --  -----------
630          45   220120   310450   12.34  1620  VideoEditor
555          38   180300   245600    6.25  3052  chrome
421          22     9860    21560    2.15  2040  explorer
310          20     8920    26432    1.90  1824  svchost
265          23    91200   121300    1.35  7896  chrome
```

Shows top 5 processes by CPU usage.

Top Processes by Memory

```
PS C:\> Get-Process | Sort-Object PM -Descending | Select-Object -First 5
Handles  NPM(K)    PM(K)    WS(K)  CPU(s)    Id  ProcessName
-------  ------    -----    -----  ------    --  -----------
630          45   220120   310450   12.34  1620  VideoEditor
398          28   120560   152200    1.45  3052  chrome
276          24    95400   128540    0.98  7896  chrome
265          23    91200   121300    0.75  8120  chrome
421          22     9860    21560    2.15  2040  explorer
```

Shows top 5 processes by memory usage.

Useful Properties

Property	Description	Example
Id	Process ID (PID)	3052
ProcessName	Name of the Process	chrome
CPU	CPU usage (in seconds)	1.45
PM	Memory usage (Private Memory in KB)	120560
WS	Working Set memory (in KB)	152200
StartTime	Time when process started	25-05-2025 10:15:32
Path	Path of the executable	C:\Program Files\...\chrome.exe

Custom Output Example

```
PS C:\> Get-Process | Select-Object ProcessName, Id, CPU,
     @(Name='Memory(MB)';Expression={[math]::Round($.PM/MB,2)}},StartTime

ProcessName     Id    CPU   Memory(MB)    StartTime
-----------     --    ---   ----------    ---------
chrome        3052   1.45   117.73        25-05-2025 10:15:32
explorer      2040   2.15     9.62        25-05-2025 08:58:10
svchost       1824   1.90     8.71        25-05-2025 09:58:05
```

Customize output to show only what you need.

Include Process Path

```
PS C:\> Get-Process chrome | Select-Object Id, ProcessName, Path

  Id  ProcessName   Path
  --  -----------   ----
3052  chrome        C:\Program Files\Google\Chrome\Application\chrome.exe
7896  chrome        C:\Program Files\Google\Chrome\Application\chrome.exe
```

Shows path of the process executable.

Pro Tips

- Use Tab for auto-completion.
- Use wildcards * to search multiple processes.
- Sort results using Sort-Object.
- Combine with Where-Object for advanced filtering.
- Use `Get-Help Get-Process -Full` for complete help.

Remember

- PID (Id) is unique for every running process.
  You can stop a process using: `Stop-Process -Id <PID>`

Get-Process = Monitor • Analyze • Manage Your Processes

Tip: Use Get-Process to keep an eye on system performance!

Process Management

Manage Processes in PowerShell

Get-Process	Start-Process	Stop-Process	Wait-Process	Kill Process
View running processes	Start a new process	Gracefully stop a process	Wait for a process to finish	Forcefully kill a process
View	Start	Stop	Wait	Kill

What is Process Management?

Process management in PowerShell allows you to start, stop, wait and monitor processes running on your local or remote computer.

Use these cmdlets to control and automate processes efficiently.

Common Workflow

Get-Process (Find Process) → Start-Process (Start New) → Wait-Process (Wait/Monitor) → Stop-Process (Stop Gracefully) → Stop-Process -Force (Kill if Needed)

1. Get-Process – View Processes

```
Get-Process

# By Name
Get-Process -Name chrome

# By ID (PID)
Get-Process -Id 1234

# Get top 5 by CPU
Get-Process | Sort-Object CPU -Descending | Select-Object -First 5
...
```

Property	Description
Id	Process ID (PID)
ProcessName	Name of the process
CPU	CPU usage (seconds)
WorkingSet	Memory usage (in KB)
StartTime	Time when started
Path	Path of the executable

Use Get-Process to list and filter running processes.

2. Start-Process – Start a New Process

```
# Start application
Start-Process notepad.exe

# Start with argument
Start-Process chrome.exe -ArgumentList "https://google.com"

# Start and open file
Start-Process "C:\Windows\System32\mspaint.exe"

# Start minimized
Start-Process powershell.exe -WindowStyle Minimized
```

Use Start-Process to launch applications or open files.

Stop-Process – Stop Processes

```
# Stop by Name
Stop-Process -Name notepad

# Stop by ID
Stop-Process -Id 1234

# Stop multiple processes
Get-Process chrome | Stop-Process

# Force stop (Kill)
Stop-Process -Name chrome -Force
```

Notes

- Stop-Process tries to close the process gracefully.
- Use -Force to kill the process if it does not respond.

Use Stop-Process to stop processes gracefully.

4. Wait-Process – Wait for a Process

```
# Wait for process by name
Wait-Process -Name notepad
# Wait for process by ID
Wait-Process -Id 1234
# Wait for multiple processes
Wait-Process -Name "chrome, notepad"
# Wait for process to exit and then run next command
Start-Process notepad; Wait-Process -Name notepad;
 Write-Host "Notepad has closed."
```

Why Wait?

- Pauses the script until the process closes.
- Useful for automation and sequencing tasks.

Use Wait-Process to pause execution until the process exits.

5. Get-Process with Filters & Sorting

```
# Filter by CPU usage
Get-Process | Where-Object { $_.CPU -gt 10 }
# Sort by memory usage
Get-Process | Sort-Object WorkingSet -Descending
# Top 10 by memory
Get-Process | Sort-Object WorkingSet -Descending | Select-Object -First 10
# Group by process name and get count
Get-Process | Group-Object ProcessName |
 Select-Object Name, Count
```

6. Process Management Examples

Task	Command
Open Notepad	Start-Process notepad.exe
Open Chrome	Start-Process chrome.exe -ArgumentList "https://google.com"
Stop Notepad	Stop-Process -Name notepad
Force Stop Chrome	Stop-Process -Name chrome -Force
Wait for Notepad to Close	Wait-Process -Name notepad
Get All Processes	Get-Process
Get Process by ID	Get-Process -Id 1234

Process States

Running (Process is actively running.) → Start (Start a new process.) → Stop (Request to end process gracefully.) → Wait (Wait until the process exits.) → Kill (Forcefully end the process.)

Pro Tips

- Use Get-Process | Format-Table -AutoSize for clean output.
- Use -PassThru with Stop-Process to confirm stopped processes.
- Combine with Where-Object for advanced filtering.
- Use Stop-Process -Force only when necessary.

Remember

Get-Process = View | Start-Process = Start

Stop-Process = Stop (Graceful) | -Force = Kill

Wait-Process = Wait for Exit

Manage processes like a pro with PowerShell! 🚀

Get-Process • Start-Process • Stop-Process • Wait-Process • Automation Made Easy

Services

Manage Windows Services with PowerShell

Get-Service	Start-Service	Stop-Service	Restart-Service	Set-Service
View services	Start a service	Stop a service	Restart a service	Change properties
List	Start	Stop	Restart	Configure

What are Services?

Services are background programs that run without user interaction. PowerShell makes it easy to view, start, stop, restart and configure Windows services.

Basic Syntax

Get-Service	Get list of services
Start-Service	Start one or more services
Stop-Service	Stop one or more services
Restart-Service	Restart one or more services
Set-Service	Change service properties

1. Get-Service – View Services

```
PS C:\> Get-Service | Select-Object Status, Name, DisplayName, StartType
Status     Name          DisplayName                                  StartType
------     ----          -----------                                  ---------
Running    Spooler       Print Spooler                                Automatic
Running    wuauserv      Windows Update                               Automatic (Delayed Start)
Stopped    WSearch       Windows Search                               Automatic (Delayed Start)
Running    WinRM         Windows Remote Management (WS-Management)    Automatic
Stopped    DiagTrack     Connected User Experiences and Telemetry     Manual
...
```

Shows service status, name, display name and startup type.

Service Status

Status	Description
Running	The service is running.
Stopped	The service is stopped.
Paused	The service is paused.
Stopped (Error)	The service is stopped due to an error.
Start Pending	The service is starting.
Stop Pending	The service is stopping.

Quick Overview

- Get list of services
- Start, Stop, Restart services
- Change startup type
- Set service properties
- Filter and find services easily

2. Common Service Actions

Start a Service

```
PS C:\> Start-Service -Name Spooler
PS C:\> Start-Service -Name wuauserv
```

Starts the specified service.

Stop a Service

```
PS C:\> Stop-Service -Name Spooler
PS C:\> Stop-Service -Name wuauserv
```

Stops the specified service gracefully.

Restart a Service

```
PS C:\> Restart-Service -Name Spooler
PS C:\> Restart-Service -Name wuauserv
```

Restarts the specified service.

Force Stop a Service (Kill)

```
PS C:\> Stop-Service -Name Spooler -Force
```

Forcibly stops the service immediately.

Note

- Use -Force only if the service cannot be stopped normally.
- Some critical services cannot be stopped.

3. Filter and Find Services

Command	Description
Get-Service -Name "wuauserv"	Get service by exact name
Get-Service -DisplayName "Windows Update*"	Get by display name (wildcard)
Get-Service | Where-Object {$_.Status -eq 'Running'}	Get running services
Get-Service | Where-Object {$_.Status -eq 'Stopped'}	Get stopped services
Get-Service | Sort-Object DisplayName	Sort services by display name
Get-Service | Select-Object Name, Status, StartType	Select specific properties

4. Set Service Startup Type (Set-Service)

Startup Type	Description
Automatic	Starts automatically with Windows
Automatic (Delayed Start)	Starts automatically after boot delay
Manual	Starts manually by user or app
Disabled	Cannot be started

```
PS C:\> Set-Service -Name Spooler -StartupType Automatic
PS C:\> Set-Service -Name wuauserv -StartupType Manual
```

Changes the startup type of a service.

5. Set Service Properties

```
PS C:\> Set-Service -Name Spooler -DisplayName "My Print Spooler"
PS C:\> Set-Service -Name Spooler -Description "Handles print jobs"
PS C:\> Set-Service -Name Spooler -StartupType Automatic
```

Change display name, description and other propertinn.

Get Help

```
PS C:\> Get-Help Get-Service -Examples
```

Get examples and detailed help.

Important Services Examples

Spooler	Print Spooler
wuauserv	Windows Update
WinRM	Windows Remote Management
WSearch	Windows Search
bits	Background Intelligent Transfer Service

Pro Tips

- Use Get-Service without parameters to list all services.
- Use Where-Object for advanced filtering.
- Always check service dependencies before stopping critical services.
- Use Restart-Service for applying changes to a service.

Remember

Get-Service = View | Start-Service = Start
Stop-Service = Stop | Restart-Service = Restart
Set-Service = Configure

Service Management Workflow

Get-Service (View Services) → Start-Service (Start Service) → Stop-Service (Stop Service) → Restart-Service (Restart Service) → Set-Service (Configure Service)

PowerShell Services – Control • Monitor • Automate

Tip: Use Get-Service daily to keep your system healthy!

Files & Folders

Manage Files and Folders in PowerShell

Get-ChildItem	New-Item	Remove-Item	Copy-Item	Move-Item	Rename-Item
List files and folders	Create new file or folder	Delete file or folder	Copy file or folder	Move file or folder	Rename file or folder

What are Files & Folders?

PowerShell provides powerful commands to work with files and folders. You can list, create, copy, move, rename and delete them easily.

Basic Syntax

`Get-ChildItem`	List files and folders
`New-Item`	Create new file or folder
`Remove-Item`	Delete file or folder
`Copy-Item`	Copy file or folder
`Move-Item`	Move file or folder
`Rename-Item`	Rename file or folder

Quick Overview

- List files and folders
- Create new files or folders
- Delete unwanted items
- Copy or move files and folders
- Rename items
- Work with files using paths and filters

1. Get-ChildItem – List Files and Folders

```
PS C:\Users\Lovejeet> Get-ChildItem
Mode          LastWriteTime          Length  Name
----          -------------          ------  ----
d----    25-05-2025    10:15  AM             Documents
d----    24-05-2025    09:40  AM             Downloads
-a---    25-05-2025    10:12  AM       1234  notes.txt
-a---    25-05-2025    10:10  AM      52248  report.pdf

PS C:\Users\Lovejeet> Get-ChildItem -Path C:\Data -Filter *.txt

Mode          LastWriteTime          Length  Name
----          -------------          ------  ----
-a---    25-05-2025    09:50  AM       2048  file1.txt
-a---    25-05-2025    09:55  AM        512  file2.txt
```

Lists files and folders in the current location or specified path.

2. Create New Files and Folders (New-Item)

```
PS C:\Users\Lovejeet> New-Item -Path C:\Data -Name "MyFolder" -ItemType Directory

    Directory: C:\Data

Mode            LastWriteTime        Name
----            -------------        ----
d-----       25-05-2025   10:20 AM    MyFolder

PS C:\Users\Lovejeet> New-Item -Path C:\Data\hello.txt -ItemType File -Value "Hello PowerShell!"

    Directory: C:\Data

Mode            LastWriteTime        Length  Name
----            -------------        ------  ----
-a----       25-05-2025   10:21 AM       17  hello.txt
```

Creates a new file or folder.

ItemType Options

Directory	→ Folder
File	→ File
SymbolicLink	→ Shortcut
HardLink	→ Hard Link
Junction	→ Junction

3. Remove-Item – Delete Files and Folders

```
# Delete a file
PS C:\Users\Lovejeet> Remove-Item -Path C:\Data\hello.txt

# Delete a folder
PS C:\Users\Lovejeet> Remove-Item -Path C:\Data\MyFolder -Recurse

# Force delete (without confirmation)
PS C:\Users\Lovejeet> Remove-Item -Path C:\Data\old.txt -Force
```

Deletes files or folders. Use -Recurse for folders and -Force to skip confirmation.

4. Copy-Item – Copy Files and Folders

```
# Dobye a file
PS C:\Users\Lovejeet> Copy-Item -Path C:\Data\notes.txt -Destination C:\Backup\

# Dobye a folder
PS C:\Users\Lovejeet> Copy-Item -Path C:\Data\MyFolder -Destination C:\Backup\ -Recurse

# Fopy with new name
PS C:\Users\Lovejeet> Copy-Item -Path C:\Data\report.pdf -Destination C:\Backup\report_copy.pdf
```

Copies files or folders to a new location.

5. Move-Item – Move Files and Folders

```
# Move a file
PS C:\Users\Lovejeet> Move-Item -Path C:\Data\notes.txt -Destination C:\Archive\

# Move and rename
PS C:\Users\Lovejeet> Move-Item -Path C:\Data\report.pdf -Destination C:\Archive\final_report.pdf

# Move a folder
PS C:\Users\Lovejeet> Move-Item -Path C:\Data\MyFolder -Destination C:\Projects\
```

Moves files or folders to a new location.

6. Rename-Item – Rename Files and Folders

```
# Rename a file
PS C:\Users\Lovejeet> Rename-Item
-Path C:\Data\old.txt -NewName new.txt

# Rename a folder
PS C:\Users\Lovejeet> Rename-Item
-Path C:\Data\OldFolder -NewName NewFolder
```

Renames files or folders.

Useful Parameters

Parameter	Description
-Path	Specify the path
-Filter	Filter files (e.g. *.txt)
-Recurse	Include all subfolders
-Force	Force operation (skip confirmation)
-ErrorAction	Set error handling behavior

Examples in One Table

Task	Command Example	Description
List all items	Get-ChildItem	List files and folders
List txt files	Get-ChildItem -Filter *.txt	List only .txt files
Create folder	New-Item -Path C:\Data -Name Test -ItemType Directory	Create a new folder
Create file	New-Item -Path C:\Data\hello.txt -ItemType File	Create a new file
Delete file	Remove-Item -Path C:\Data\hello.txt	Delete a file
Copy file	Copy-Item -Path C:\Data\a.txt -Destination C:\Backup\	Copy file to another location
Move file	Move-Item -Path C:\Data\a.txt -Destination C:\Archive\	Move file to another location
Rename file	Rename-Item -Path C:\Data\a.txt -NewName b.txt	Rename a file

Pro Tips

- Use Tab key for auto-completion.
- Use -Recurse carefully while deleting or moving folders.
- Use -WhatIf with commands to preview changes.
- Use -Force only when necessary.
- Combine wildcards (*) with -Filter for powerful searches.

Remember

Always double-check the path before deleting or moving files!

Common File Wildcards

Wildcard	Matches
*	All files and folders
*.txt	All text files
*.docx	All Word documents
*.png	All PNG images
file*	Files starting with 'file'
2025	Files containing '2025'

PowerShell **Files & Folders** – List • Create • Copy • Move • Rename • Delete

Tip: Practice these commands daily to become a PowerShell Pro!

Navigation

Navigate File System in PowerShell

Get-Location	Set-Location	Push-Location	Pop-Location	Get-ChildItem
Show current location	Change current location	Save current location to stack	Restore previous location	List items in current location

Navigation Concepts

Path	Location of files and folders in the file system.
Directory	A folder that can contain files and other folders.
Drive	Storage device like C:, D:, E: etc.

Basic Syntax

`Get-Location`	Get current location
`Set-Location`	Change current location
`Push-Location`	Save current location
`Pop-Location`	Restore previous location
`Get-ChildItem`	List items in current location

Quick Overview

- View your current location
- Change to another path or drive
- Save and restore locations
- List files and folders
- Work with relative and absolute paths

1. Get-Location – Show Current Location

```
PS C:\Users\Lovejeet> Get-Location

Path
----
C:\Users\Lovejeet
```

Shows the current working directory.

Navigation Commands at a Glance

Command	Description	Example
Get-Location	Get current location	Get-Location
Set-Location	Change current location	Set-Location C:\Windows
Push-Location	Save current location to stack	Push-Location C:\Temp
Pop-Location	Restore previous location	Pop-Location
Get-ChildItem	List files and folders	Get-ChildItem

2. Set-Location – Change Location

```
PS C:\Users\Lovejeet> Set-Location C:\Windows
PS C:\Windows> Get-Location

Path
----
C:\Windows
```

Change to a specific path or drive.

3. Push-Location – Save Location

```
PS C:\Windows> Push-Location C:\Temp
PS C:\Temp> Get-Location

Path
----
C:\Temp
```

Saves current location and moves to new location.

4. Pop-Location – Restore Previous

```
PS C:\Temp> Pop-Location
PS C:\Windows> Get-Location

Path
----
C:\Windows
```

Restores the previous saved location.

5. Get-ChildItem – List Contents

```
PS C:\Windows> Get-ChildItem
Mode          LastWriteTime       Length  Name
----          -------------       ------  ----
d-----    25-05-2025  09:30 AM            System32
d-----    25-05-2025  09:30 AM            Temp
-a----    25-05-2025  08:15 AM     17408  win.ini
-a----    25-05-2025  08:15 AM     79872  setupact.log
```

Lists files and folders in the current location.

Common Paths Examples

Path Type	Example	Description
Absolute Path	C:\Windows\System32	Full path from drive root
Relative Path	.\Documents	Relative to current location
Parent Directory	..	Go up one level
Root Directory	C:\	Root of C drive
Another Drive	D:\Projects	Switch to another drive

Drives in Windows

Drive	Description
C:	System Drive (OS installed)
D:	Data Drive
E:	Removable / External Drive
Network Drive	\\Server\Share

Get-ChildItem with Details

```
PS C:\Windows> Get-ChildItem -Force
Mode          LastWriteTime       Length  Name
----          -------------       ------  ----
d--h--    25-05-2025  09:31 AM            Fonts
d-----    25-05-2025  09:30 AM            Logs
-a-h--    25-05-2025  07:50 AM      1234  hidden.txt
```

Use -Force to show hidden items too.

Pro Tips

- Use Tab key for auto-completion.
- Use .\ for current directory, ..\ to go up.
- Use Push-Location and Pop-Location to move safely.
- Use Get-ChildItem -Recurse to list all subfolders.

Remember

Your current location (also called working directory) decides where PowerShell looks for files.

Note

Be careful while using Set-Location on system folders. Always double-check the path.

Navigate with PowerShell like a Pro! | Get-Location • Set-Location • Push-Location • Pop-Location • Get-ChildItem Tip: Clear-Host to clear the screen and stay focused!

File Content

Read, Write & Manage File Content in PowerShell

Get-Content	Set-Content	Add-Content	Clear-Content
Read File	Write File	Append Text	Clear File
Read	Write	Append	Clear

What is File Content?

PowerShell lets you read, write, append and clear the content of text files easily using simple commands.

Basic Syntax

Get-Content	Read content from a file
Set-Content	Write content to a file (overwrite)
Add-Content	Add content to the end of a file
Clear-Content	Clear all content from a file

Quick Overview

- Get-Content → Read a file
- Set-Content → Write/Overwrite a file
- Add-Content → Append to a file
- Clear-Content → Clear a file

1. Get-Content – Read File Content

```
PS C:\Data> Get-Content .\notes.txt
PowerShell is awesome!
It helps in automation.
Learn PowerShell every day.
```

Reads the content of a file.

2. Set-Content – Write/Overwrite File

```
PS C:\Data> Set-Content .\notes.txt "Hello PowerShell!"

PS C:\Data> Get-Content .\notes.txt
Hello PowerShell!
```

Writes content to a file and overwrites if file exists.

3. Add-Content – Append to File

```
PS C:\Data> Add-Content .\notes.txt "Learning is fun!"
PS C:\Data> Get-Content .\notes.txt
Hello PowerShell!
Learning is fun!
```

Adds content to the end of the file.

4. Clear-Content – Clear File

```
PS C:\Data> Clear-Content .\notes.txt

PS C:\Data> Get-Content .\notes.txt
```

Removes all content from the file.

Examples in Action

Step	Command	Note
1 Create Sample File	`PS C:\Data> Set-Content .\demo.txt "Line 1"`	# Create and write initial content
2 Read File	`PS C:\Data> Get-Content .\demo.txt` `Line 1`	# Read content from file

3 Append Content	`PS C:\Data> Add-Content .\demo.txt "Line 2"`
	`PS C:\Data> Get-Content .\demo.txt`
	`Line 1`
	`Line 2`
		# Append new line to file

4 Overwrite File	`PS C:\Data> Set-Content .\demo.txt "New Line 1"`
	`PS C:\Data> Get-Content .\demo.txt`
	`New Line 1`
		# Overwrite existing content

5 Clear File	`PS C:\Data> Clear-Content .\demo.txt` `PS C:\Data> Get-Content .\demo.txt`	# Clear all content

Read File with Options (Get-Content)

Command	Description	Example
Get-Content file.txt	Read all content	Get-Content .\file.txt
Get-Content file.txt -TotalCount 5	Read first 5 lines	Get-Content .\file.txt -TotalCount 5
Get-Content file.txt -Tail 5	Read last 5 lines	Get-Content .\file.txt -Tail 5
Get-Content file.txt -Raw	Read as single string	Get-Content .\file.txt -Raw

Write File with Options (Set-Content)

Command	Description	Example
Set-Content file.txt "Text"	Write/overwrite text	Set-Content .\file.txt "Hello"
Set-Content file.txt -NoNewline "Text"	Write without new line	Set-Content .\file.txt -NoNewline "Hello"
Set-Content file.txt -Encoding UTF8 "Text"	Write with encoding	Set-Content .\file.txt -Encoding UTF8 "Hello"

File Content Workflow

Read (Get-Content) → Modify (Optional) → Append (Add-Content) → Write (Set-Content) → Clear (Clear-Content)

Real World Example

```
PS C:\Data> Get-Date | Out-String | Add-Content .\log.txt
  # Add current date to log file
PS C:\Data> Get-Content .\log.txt                    # View log
 25-05-2025  09:30:15
PS C:\Data> Set-Content .\log.txt "Log Started..."  # Start new log (overwrite)
PS C:\Data> Add-Content .\log.txt "App is running"  # Append new log entry
PS C:\Data> Get-Content .\log.txt
Log Started...
App is running
```

Pro Tips

- Use Get-Content -Raw for reading entire file as one string.
- Set-Content overwrites the file. Use Add-Content to append.
- Always specify -Encoding UTF8 for better compatibility.
- Use Clear-Content carefully, it removes everything!

Common File Content Commands at a Glance

Command	Purpose
Get-Content	Read content from a file
Set-Content	Write/overwrite content to a file
Add-Content	Append content to a file
Clear-Content	Clear all content from a file

Remember

- Read → Get-Content
- Write/Overwrite → Set-Content
- Append → Add-Content
- Clear → Clear-Content

Note

These commands work best with text files (.txt, .log, .csv, .ini, .conf, etc.).

Not recommended for binary files (like .exe, .jpg, .mp4).

Sample File (notes.txt)

```
Hello PowerShell!
PowerShell makes system management easy.
Automate tasks and save time.
Keep learning and stay productive!
```

Supported File Types (Text Based)

- → .txt Text files
- → .log Log files
- → .csv CSV files
- → .ini Config files
- → .conf Configuration files

Master File Content in PowerShell

Get-Content • Set-Content • Add-Content • Clear-Content

Read • Write • Append • Clear

File Search

Search Files Easily in PowerShell

Get-ChildItem	-Recurse	-Filter	-Include	-Exclude	Where-Object
List files & folders	Search in all subfolders	Filter by pattern	Include specific files	Exclude specific files	Filter with conditions

What is File Search?

PowerShell provides powerful options to search files by name, extension, size, date and more using Get-ChildItem and Where-Object.

Basic Syntax

Get-ChildItem	List files and folders
-Recurse	Search in all subfolders
-Filter	Filter using wildcard pattern
-Include	Include only specific files
-Exclude	Exclude specific files
Where-Object	Filter using conditions

Quick Overview

- Use Get-ChildItem to find files
- Add -Recurse to search in subfolders
- Use -Filter, -Include, -Exclude for patterns
- Use Where-Object for advanced filtering
- Combine options for powerful search

1. Basic File Search

```
PS C:\> Get-ChildItem -Path C:\Data

   Directory: C:\Data

Mode          LastWriteTime       Length Name
-a----    25-05-2025   09:30 AM   1234   notes.txt
-a----    25-05-2025   09:31 AM   2048   report.docx
-a----    25-05-2025   09:32 AM    512   data.csv
-d----    25-05-2025   09:33 AM          Projects
```

Lists files and folders in the specified path.

2. Search in All Subfolders (-Recurse)

```
PS C:\> Get-ChildItem -Path C:\Data -Recurse

   Directory: C:\Data\Projects\2025

Mode              LastWriteTime      Length Name
-a----      25-05-2025   10:10 AM    3021   plan.txt
-a----      25-05-2025   10:12 AM    5320   budget.xlsx

   Directory: C:\Data\Projects\2025\Reports

-a----      25-05-2025   10:15 AM    4120   summary.pdf
```

Searches files in the current folder and all subfolders.

3. Filter Files by Pattern (-Filter)

```
PS C:\> Get-ChildItem -Path C:\Data -Filter *.txt

   Directory: C:\Data

Mode             LastWriteTime          Length  Name
----             -------------          ------  ----
-a----      25-05-2025   09:30 AM       1234    notes.txt
-a----      25-05-2025   10:10 AM       3021    plan.txt
```

Finds files that match the pattern (e.g., *.txt).

4. Include Specific Files (-Include)

```
PS C:\> Get-ChildItem -Path C:\Data -Recurse -Include *.txt,*.log

   Directory: C:\Data

Mode          LastWriteTime          Length  Name
----          -------------          ------  ----
-a----    25-05-2025   09:30 AM      1234    notes.txt
-a----    25-05-2025   10:10 AM      3021    plan.txt
-a----    25-05-2025   11:20 AM      2048    app.log
```

Includes only files that match the given extensions.

5. Exclude Files (-Exclude)

```
PS C:\> Get-ChildItem -Path C:\Data -Recurse -Exclude *.log,*.tmp

   Directory: C:\Data

Mode          LastWriteTime          Length  Name
----          -------------          ------  ----
-a----    25-05-2025   09:30 AM      1234    notes.txt
-a----    25-05-2025   09:31 AM      2048    report.docx
-a----    25-05-2025   10:12 AM      5320    budget.xlsx
```

Excludes files that match the given extensions.

6. Advanced Search with Where-Object

```
PS C:\> Get-ChildItem -Path C:\Data -Recurse | Where-Object
{ $_.Length -gt 2000 }

   Directory: C:\Data\Projects\2025

Mode          LastWriteTime          Length  Name
----          -------------          ------  ----
-a----    25-05-2025   10:12 AM      5320    budget.xlsx
-a----    25-05-2025   10:15 AM      4120    summary.pdf
```

Finds files larger than 2000 bytes using Where-Object.

Useful Wildcards

*	⇒	Any number of characters
?	⇒	Single character
*.txt	⇒	All txt files
report.*	⇒	report with any extension
data??.csv	⇒	data01.csv, data02.csv

7. Real World Examples

```
Get-ChildItem -Path C:\Data -Recurse -Filter *.pdf                                   # Find all PDF files
Get-ChildItem -Path C:\Data -Recurse -Include *.txt,*.md                             # Include only txt and md
Get-ChildItem -Path C:\Data -Recurse -Exclude *.log,*.tmp                            # Exclude log and tmp
Get-ChildItem -Path C:\Data -Recurse | Where-Object { $_.LastWriteTime -gt (Get-Date).AddDays(-7) } # Files modified in last 7 days
Get-ChildItem -Path C:\Data -Recurse | Where-Object { $_.Name -like "*report*" }      # Files with 'report' in name
Get-ChildItem -Path C:\Data -File | Sort-Object Length -Descending | Select-Object -First 5   # Top 5 largest files
```

Search Options Summary

Option	Use For	Example
-Recurse	Search in all subfolders	Get-ChildItem -Recurse
-Filter	Simple pattern filtering	Get-ChildItem -Filter *.txt
-Include	Include specific files	Get-ChildItem -Include *.txt,*.log
-Exclude	Exclude specific files	Get-ChildItem -Exclude *.temp
Where-Object	Advanced conditions	Get-ChildItem | Where-Object { $_.Length -gt 1MB }

Pro Tips

- Always use -Recurse carefully in large folders.
- Use -Filter for simple name patterns (faster).
- Use Where-Object for size, date, or content filters.
- Combine options for powerful and precise search.

Remember

Get-ChildItem is your best friend for searching files in PowerShell!

Search Smarter, Save Time! ⚡ Get-ChildItem • -Recurse • -Filter • -Include • -Exclude • Where-Object Find Any File, Anywhere!

Copy & Move

Copy or Move Files & Folders in PowerShell

Copy-Item	Move-Item	-Recurse	-Force	Destination	Source
Copy files or folders	Move files or folders	Include all subfolders	Overwrite & suppress errors	Where items go	Where items come from

What is Copy & Move?

Use Copy-Item to duplicate items to a new location and Move-Item to relocate items to a new location.

Basic Syntax

```
Copy-Item
Copy-Item <Source> <Destination> [Options]

Move-Item
Move-Item <Source> <Destination> [Options]
```

Quick Overview

- Copy-Item duplicates files or folders.
- Move-Item moves files or folders.
- Use -Recurse to include subfolders.
- Use -Force to overwrite and ignore errors.
- Source = where items come from.
- Destination = where items go.

1. Copy Files and Folders (Copy-Item)

```
# Copy a file
Copy-Item "C:\Data\notes.txt" "C:\Backup\notes.txt"

# Copy a folder
Copy-Item "C:\Data\Project" "D:\Projects\Project" -Recurse

# Copy and overwrite if exists
Copy-Item "C:\Data\report.pdf" "D:\Reports\report.pdf" -Force
```

Duplicates files or folders to the destination.

2. Move Files and Folders (Move-Item)

```
# Move a file
Move-Item "C:\Data\notes.txt" "D:\Archive\notes.txt"

# Move a folder
Move-Item "C:\Data\Project" "D:\Projects\Project" -Recurse

# Move and overwrite if exists
Move-Item "C:\Data\old_report.pdf" "D:\Reports\report.pdf" -Force
```

Moves files or folders to the destination.

Options Explained

- **-Recurse** Include all subfolders and files.
- **-Force** Overwrite read-only files and suppress errors.
- **-WhatIf** Shows what would happen without making changes.
- **-Confirm** Asks for confirmation before overwriting.

3. Copy & Move with -Recurse

```
# Copy folder with all subfolders and files
Copy-Item "C:\Data\Project" "E:\Backup\Project" -Recurse

# Move folder with all subfolders and files
Move-Item "C:\Data\Project" "E:\Archive\Project" -Recurse
```

-Recurse includes everything inside the folder.

4. Copy & Move with -Force

```
# Force copy (overwrite if exists)
Copy-Item "C:\Data\report.pdf" "D:\Reports\report.pdf" -Force

# Force move (overwrite if exists)
Move-Item "C:\Data\old.pdf" "D:\Reports\old.pdf" -Force
```

-Force overwrites files and ignores read-only errors.

5. Use -WhatIf to Preview

```
# Preview copy (no changes made)
Copy-Item "C:\Data\file.txt" "D:\Test\file.txt" -WhatIf

# Preview move (no changes made)
Move-Item "C:\Data\file.txt" "D:\Test\file.txt" -WhatIf
```

Great for safe testing before actual execution.

6. Use -Confirm for Safety

```
# Confirm before overwriting
Copy-Item "C:\Data\file.txt" "D:\file.txt" -Confirm

# Confirm before moving
Move-Item "C:\Data\file.txt" "D:\file.txt" -Confirm
```

Asks for confirmation before each action.

Common Parameters

Parameter	Description
-Recurse	Include all subfolders and files
-Force	Overwrite read-only files, suppress errors
-WhatIf	Show what would happen (no changes made)
-Confirm	Ask for confirmation before overwriting
-ErrorAction	Set error handling behavior

Examples in One Table

Task	Command (Copy-Item)	Command (Move-Item)
Copy file	Copy-Item "a.txt" "D:\a.txt"	Move-Item "a.txt" "D:\a.txt"
Copy folder	Copy-Item "Folder" "D:\Folder" -Recurse	Move-Item "Folder" "D:\Folder" -Recurse
Copy with overwrite	Copy-Item "file.txt" "D:\file.txt" -Force	Move-Item "file.txt" "D:\file.txt" -Force
Preview copy	Copy-Item "file.txt" "D:\" -WhatIf	Move-Item "file.txt" "D:\" -WhatIf
Confirm copy	Copy-Item "file.txt" "D:\" -Confirm	Move-Item "file.txt" "D:\" -Confirm

Real World Example

```
# Backup important data
Copy-Item "C:\Work\" "D:\Backup\Work_$(Get-Date -Format yyyyMMdd)" -Recurse

# Move old logs to archive
Move-Item "C:\Logs\*.log" "D:\Archive\Logs\" -Force

Write-Host "Backup and cleanup completed successfully!"
```

Use in scripts for backup, cleanup, and organization.

Pro Tips

- Always use -Recurse for folders.
- Use -Force carefully.
- Use -WhatIf while testing scripts.
- Combine with Get-ChildItem for bulk operations.

Remember

Copy-Item = Duplicate
Move-Item = Relocate
Source ⟶ where from
Destination ⟶ where to

Note

Be careful with -Force. It can overwrite important files without warning.

Supported Items

Files · Folders · Subfolders · Any Item

Copy & Move Workflow

Source (Select item) → Choose Action (Copy-Item or Move-Item) → Add Options (-Recurse, -Force, -WhatIf, -Confirm) → Destination (Specify target location) → Done (Item copied or moved)

Quick Command Reference

```
Copy-Item source dest              Copy a file or folder
Copy-Item source dest -Recurse     Copy folder with all contents
Copy-Item source dest -Force       Copy and overwrite
Move-Item source dest              Move a file or folder
Move-Item source dest -Recurse     Move folder with all contents
Move-Item source dest -Force       Move and overwrite
```

Copy & Move in PowerShell – Fast, Flexible & Powerful! 🚀 Copy-Item • Move-Item • -Recurse • -Force • Smart File Management Organize Today, Save Time Tomorrow! ⏰

Remove Items

Delete Files & Folders in PowerShell

Remove-Item	-Recurse	-Force	-Confirm	File	Folder
Delete files or folders	Remove all subfolders & files	Force remove (read-only, hidden)	Ask confirmation before deleting	Delete a file	Delete a folder

What is Remove-Item?

Remove-Item deletes files and folders. Use advanced options to safely and efficiently remove items.

Basic Syntax

```
Remove-Item  <Path> [Options]
Remove-Item  <Path> -Recurse
Remove-Item  <Path> -Force
Remove-Item  <Path> -Confirm
```

Quick Overview

- Delete files and folders.
- Use -Recurse for folders with content.
- Use -Force to remove read-only/hidden items.
- Use -Confirm to avoid accidental deletion.
- Be careful! Deleted items cannot be recovered.

1. Delete a File

```
# Delete a file
Remove-Item "C:\Data\notes.txt"
# OR
Remove-Item .\notes.txt
```

Deletes the specified file.

2. Delete a Folder (Empty)

```
# Delete an empty folder
Remove-Item "C:\Temp\OldFolder"
```

Deletes the specified empty folder.

3. Delete Folder with Contents (-Recurse)

```
# Delete folder and all its content
Remove-Item "C:\Projects\2025" -Recurse
```

Deletes folder and all subfolders/files.

4. Force Delete (Read-only / Hidden)

```
# Force delete read-only file
Remove-Item "C:\Data\readme.txt" -Force
```

Deletes read-only, hidden or system items.

5. Force + Recurse

```
# Force delete folder with all content
Remove-Item "C:\Backup" -Recurse -Force
```

Deletes folder and all contents, ignoring errors.

6. Confirm Before Deleting

```
# Ask confirmation before deleting
Remove-Item "C:\Data\report.docx" -Confirm
```

Prompts for confirmation before deletion.

7. Delete Multiple Items

```
# Delete multiple files
Remove-Item "C:\Data\*.tmp" -Force
```

Deletes all .tmp files in the folder.

8. Delete by Filter (Advanced)

```
# Delete files older than 7 days
Get-ChildItem "C:\Logs" -File |
  Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-7) } |
  Remove-Item -Force
```

Deletes files older than 7 days.

Common Parameters

Parameter	Description
-Recurse	Include all subfolders and files.
-Force	Force remove. Ignore read-only or hidden items.
-Confirm	Ask for confirmation before deleting.
-WhatIf	Show what would happen without deleting.
-Path	Specify the item(s) to remove.
-ErrorAction	Set error handling behavior.

Examples in One Table

Task	Command Example
Delete file	Remove-Item "C:\Data\file.txt"
Delete empty folder	Remove-Item "C:\Temp\EmptyFolder"
Delete folder with content	Remove-Item "C:\Projects" -Recurse
Force delete read-only file	Remove-Item "C:\Data\file.txt" -Force
Force delete folder	Remove-Item "C:\Backup" -Recurse -Force
Confirm before delete	Remove-Item "C:\Data\important.docx" -Confirm
WhatIf preview	Remove-Item "C:\Data*.log" -WhatIf

WhatIf Example (Preview Mode)

```
# Preview deletion (no actual delete)
Remove-Item "C:\Data\*.log" -WhatIf
What if: Performing the operation "Remove File"
on target "C:\Data\log1.log".
What if: Performing the operation "Remove File"
on target "C:\Data\log2.log".
```

Shows what would be deleted, but does not delete.

Pro Tips

- Always use -Confirm for important files.
- Use -WhatIf first to preview large deletions.
- Use -Recurse carefully. It deletes everything inside.
- Deleted items cannot be recovered from PowerShell.

Remember

Be careful when using -Force.
It can delete read-only or hidden system files.

Common Wildcards

Wildcard	Matches
*	All files and folders
*.txt	All text files
*.log	All log files
*.docx	All Word documents
?	Single character
file??.txt	file01.txt, fileAB.txt

Caution!

Use Remove-Item carefully.
There is no recycle bin in PowerShell.
Deleted items are gone forever!

Safe Deletion Workflow

Find Items
Get-ChildItem
→
Preview
-WhatIf
→
Confirm
-Confirm
→
Delete
Remove-Item
→
Done
Items Removed

Real World Example

```
# Clean temp files older than 7 days
Get-ChildItem "C:\Temp" -File |
  Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-7) } |
  Remove-Item -Force -WhatIf
```

Preview then delete old temp files safely.

Note

Always double-check the path before running Remove-Item, especially with -Recurse and -Force.

Remove Smartly, Work Safely! Remove-Item • -Recurse • -Force • -Confirm • -WhatIf Delete Today, Keep System Clean!

PowerShell Variables

Store Data • Use Values • Make Scripts Powerful

$Name — Store text or data
$Age — Store number or value
$Path — Store file/folder path
$Files — Store file list or array
$Process — Store process information

What are Variables?

Variables are used to store data in PowerShell. You can store text, numbers, paths, files, processes and more in variables.

Basic Syntax

```
$Name  =  Value        # Variable name and value
$Name  =  Value        # Assignment
$Name                  # Use the variable
$Name -Type            # (Optional) set data type
```

Quick Overview

- Use $ to create a variable
- Assign a value with =
- Store text, numbers, paths, files, etc.
- Variables make your scripts flexible and reusable.
- You can use the same variable again and again.

1 String Variable ($Name)

```
PS C:\> $Name = "Lovejeet"
PS C:\> Write-Host "Hello $Name!"
Hello Lovejeet!
```

Used to store text (string) values.

2 Number Variable ($Age)

```
PS C:\> $Age = 30
PS C:\> Write-Host "My age is $Age"
My age is 30
```

Used to store numbers (integer or decimal).

3 Path Variable ($Path)

```
PS C:\> $Path = "C:\Windows\System32"
PS C:\> Get-ChildItem $Path
Mode          LastWriteTime          Length Name
------        -------------          ------ ----
d-----     25-05-2025  09:30 AM             drivers
```

Used to store file or folder paths.

4 Array/List Variable ($Files)

```
PS C:\> $Files = Get-ChildItem *.txt
PS C:\> $Files
file1.txt     file2.txt     file3.txt
file4.txt     file5.txt
```

Used to store multiple values (array or list).

5 Process Variable ($Process)

```
PS C:\> $Process = Get-Process -Name notepad
PS C:\> $Process | Select-Object Name, Id
Name          Id
----          --
notepad       1234
```

Used to store process information.

6 Variable with Assignment

```
PS C:\> $Message = "PowerShell"
PS C:\> Write-Host $Message
PowerShell
```

Use = to assign a value to a variable.

7 Reusing Variables

```
PS C:\> $Name = "Sardar"
PS C:\> Write-Host "Welcome $Name"
Welcome Sardar
```

You can use the same variable again and again.

8 Multiple Variables

```
PS C:\> $Name = "Sardar"
PS C:\> $Age = 25
PS C:\> $Path = "C:\Temp"
PS C:\> Write-Host "$Name is $Age years old."
Sardar is 25 years old.
```

You can create and use multiple variables.

Common Variables

Variable	Description
$Name	Stores a name or text
$Age	Stores a number
$Path	Stores a file/folder path
$Files	Stores a list of files
$Process	Stores process information

Variable Data Types

Type	Example
String (Text)	$Name = "Hello"
Integer (Number)	$Age = 25
Double (Decimal)	$Price = 10.5
Boolean (True/False)	$IsDone = $true
Array (List)	$Files = @("a.txt","b.txt")

Pro Tips

- Use meaningful variable names.
- Follow camelCase (e.g. $userName).
- Use single quotes for simple strings.
- Use double quotes when you need variables inside a string.
- Use Get-Help <command> to learn more.

Remember

Variables are case-insensitive, so $Name and $name are the same.

Common Mistakes

- Forgetting the $ sign.
- Using spaces in variable names (e.g. $my name).
- Not using quotes for text.
- Assigning wrong data type.
- Overwriting important variables.

Example Script

```
$Folder = "C:\Temp"
$Files = Get-ChildItem $Folder -Filter *.txt
$Count = $Files.Count
Write-Host "Total text files: $Count"
```

PowerShell Variables – Store Once, Use Everywhere! $Name • $Age • $Path • $Files • $Process

Tip: Use variables to make your scripts smarter and easier to manage!

Environment Variables

Access & Manage Environment Variables in PowerShell

$env:PATH	$env:USERPROFILE	$env:TEMP	$env:USERNAME
System search path for executables	Current user's profile directory	Directory for temporary files	Current user's login name

What are Environment Variables?

Environment variables are dynamic named values that affect the way processes run on your system.

1. $env:PATH

Displays the system PATH environment variable.

```
PS C:\> $env:PATH
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System
32\WindowsPowerShell\v1.0);C:\Program Files\Git\cmd;,...
```

Output: The directories listed in the PATH variable.

2. $env:USERPROFILE

Displays the current user profile directory.

```
PS C:\> $env:USERPROFILE
C:\Users\JohnDoe
```

Output: The path to the current user's profile folder.

Environment Variable Types

- **System:** Available to all users on the computer.
- **User:** Specific to the current user account.

3. $env:TEMP

Displays the directory for temporary files.

```
PS C:\> $env:TEMP
C:\Users\JohnDoe\AppData\Local\Temp
```

Output: The folder used for temporary files.

4. $env:USERNAME

Displays the current user's login name.

```
PS C:\> $env:USERNAME
JohnDoe
```

Output: The username of the currently logged-in user.

Where Are Environment Variables Stored?

System Variables
Stored in the system registry (HKEY_LOCAL_MACHINE).

User Variables
Stored in the user registry (HKEY_CURRENT_USER).

View All Environment Variables

Get all environment variables.

```
PS C:\> Get-ChildItem Env:
```

Output: Lists all environment variables and their values.

Set an Environment Variable

Create or update an environment variable.

```
PS C:\> $env:MYVAR = "Hello PowerShell"
PS C:\> $env:MYVAR
Hello PowerShell
```

Output: The variable is set and can be accessed.

Use Environment Variables

Use variables in commands and scripts.

```
PS C:\> "Temp folder is $env:TEMP"
Temp folder is C:\Users\JohnDoe\AppData\Local\Temp
```

Output: The variable value is used in the command.

Quick Tips

- Environment variables are case-insensitive.
- Use $env: to access current session variables.
- Changes to user variables require new processes to see the update.

Manage Your Environment, Automate More! ⚡ $env:PATH • $env:USERPROFILE • $env:TEMP • $env:USERNAME PowerShell Makes It Easy! 😎

Arrays & Collections

Store, Manage & Work with Multiple Values in PowerShell

Array	$List	Index	Count	ForEach	Collections	Objects	Values
@()		0, 1, 2...	.Count	ForEach-Object		{ }	
Ordered list of values	Store items in a variable	Position of item in array	Total number of items	Loop through each item	Group of objects/values	Key-Value pair data	Data stored in collections

Basic Syntax

```
$Array = @("One", "Two", "Three")
```

@() → Array literal
$Array → Variable that holds the array

1. Create an Array

```
$Fruits = @("Apple", "Banana", "Mango")

$Numbers = @(1, 2, 3, 4, 5)

$Mixed = @("Jeet", 25, $true, 5.5)
```

Creates arrays using @() syntax.

2. Access by Index

```
$Fruits = @("Apple", "Banana", "Mango")
$Fruits[0]    # Apple
$Fruits[1]    # Banana
$Fruits[2]    # Mango
$Fruits[-1]   # Last item (Mango)
```

Index starts from 0. Use -1 for last item.

3. Get Count / Length

```
$Fruits = @("Apple", "Banana", "Mango")

$Fruits.Count      # 3
$Fruits.Length     # 3

( @(1,2,3,4,5) ).Count      # 5
```

.Count and .Length return total items.

4. Loop with ForEach

```
$Fruits = @("Apple", "Banana", "Mango")

$Fruits | ForEach-Object {
   "Fruit: $_"
}
```

ForEach-Object loops through each item.

Quick Overview

- Arrays store multiple items in a single variable.
- Items are accessed using index (starting from 0).
- Use .Count to get total number of items.
- Use ForEach-Object to loop through items.
- Collections can store arrays, objects, or values.

Remember

- Index always starts at 0.
- Arrays can store different data types.
- Arrays are fixed in size, use ArrayList for dynamic size.
- Collections help manage complex data.

5. Add / Combine Arrays

```
$A = @(1,2,3)
$B = @(4,5,6)
$C = $A + $B       # Combine arrays
$C
#  Output: 1 2 3 4 5 6
```

Use + operator to combine arrays.

6. Check if Item Exists

```
$Fruits = @("Apple", "Banana", "Mango")
if($Fruits -contains "Banana"){
    "Banana found!"
}else{
    "Not found"
}
```

Use -contains to check item in array.

7. Filter / Select Items

```
$Numbers = @(1,2,3,4,5,6,7,8,9,10)
$Even = $Numbers | Where-Object { $_ % 2 -eq 0 }
$Even
# Output: 2 4 6 8 10
```

Use Where-Object to filter items.

8. Sort Items

```
$Numbers = @(5,2,9,1,3)
$Numbers | Sort-Object
# Output: 1 2 3 5 9
```

Sort-Object sorts array items.

Common Array Methods

Method	Description
.Count	Get total number of items
.Length	Get length of the array
.GetValue(index)	Get item at index
.SetValue(value, index)	Set item at index

Collections (ArrayList Example)

```
$List = New-Object System.Collections.ArrayList
$List.Add("One")    | Out-Null
$List.Add("Two")    | Out-Null
$List.Add("Three")  | Out-Null

$List.Count            # 3
```

ArrayList allows dynamic add/remove of items.

Objects (PSCustomObject Example)

```
$Person = [PSCustomObject]@{
   Name   = "Lovejeet Singh"
   Age    = 32
   City   = "Amritsar"
}
$Person
```

Objects store data in Key-Value pairs.

Array of Objects

```
$People = @()
$People += [PSCustomObject]@{Name="Jeet"; Age=32}
$People += [PSCustomObject]@{Name="Aman"; Age=25}

$People | Format-Table
```

Store multiple objects in an array.

Common Operators with Arrays

Operator	Description
+	Combine arrays
-contains	Check if item exists
|	Pipe to process items
[index]	Access item by index

Pro Tips

- Use meaningful variable names.
- Use ForEach-Object for clean loops.
- Use -contains for fast searching.
- Use ArrayList for dynamic collections.

Real World Example

Read files from a folder and display names:

```
$Files = Get-ChildItem -Path "C:\Data" -File
$Files | ForEach-Object { $_.Name }
```

Stores files in array and loops through names.

Array Workflow

Create Array @() → Access Items [index] → Check / Filter -contains / Where → Loop Items ForEach-Object → Get Results Output

Helpful Commands

Command	Description
Get-ChildItem	Get files and folders
Where-Object	Filter items
Sort-Object	Sort items
Select-Object	Select properties

Note

- Arrays are fixed size.
- ArrayList or Collections are dynamic.
  Be careful with index out of range.

Arrays make your data organized! @() • $List • Index • Count • ForEach • Collections • Objects • Values Work Smart, Automate More!

Objects & Properties

Work with PowerShell Objects Like a Pro!

Object	Properties	Methods	Get-Member	PSObject	Type	Value	Object Pipeline
Represents real world entity	Data / values of an object	Actions / functions of an object	Explore members of an object	Base object type in PowerShell	.GetType() shows object type	Actual data stored in property	Objects flow through the pipeline

What is an Object?

In PowerShell, everything is an object. Objects have Properties (data) and Methods (actions).

Quick Overview

- Objects can store data and behavior.
- Use dot notation (.Property) to access properties.
- Use Get-Member to view all properties and methods.
- Objects can be passed through pipeline.

Remember

- Objects are more powerful than text.
- Use Get-Member to explore any object.
- Properties store values.
- Methods perform actions.

Common Get-Member Types

Member Type	Description
Property	Data value of the object
Method	Action that can be performed
NoteProperty	Custom property (added)
ScriptProperty	Calculated property
AliasProperty	Alias of another property
Event	Event that object can raise

1. Create an Object (PSCustomObject)

```
$person = [PSCustomObject]@{
   Name    = "Lovejeet Singh"
   Age     = 32
   City    = "Amritsar"
   Job     = "Lab Attendant"
}
```

Creates a custom object with properties.

2. Access Properties

```
$person.Name     # Lovejeet Singh
$person.Age      # 32
$person.City     # Amritsar
$person.Job      # Lab Attendant
```

Use dot notation to access property values.

3. View All Members (Get-Member)

```
$person | Get-Member

# Lists all properties and methods
# available in the object.
```

Discover all members of the object.

4. Filter Members by Type

```
$person | Get-Member -MemberType Property
$person | Get-Member -MemberType Method
```

Filter members using -MemberType.

5. Call Methods (if any)

```
$date = Get-Date
$date.ToShortDateString() # Method call
$date.AddDays(5)       # Returns new date
```

Use methods to perform actions.

6. Object in the Pipeline

```
Get-Process | Select-Object -First 1
    | Format-List *

# Objects flow through the pipeline
# and keep their properties.
```

Pipeline passes objects, not just text.

7. Get Object Type

```
$person.GetType().FullName
# Output: System.Management.Automation.PSCustomObject

$person.GetType().Name
# Output: PSCustomObject
```

Check the type of the object.

8. Add New Property to Object

```
$person | Add-Member -NotePropertyName Email -
 NotePropertyValue "jeet@example.com"

$person.Email   # jeet@example.com
```

Add new properties dynamically.

Example Output (Get-Member)

```
   TypeName: System.Management.Automation.PSCustomObject

Name          MemberType    Definition
----          ----------    ----------
Equals        Method        bool Equals(System.Object obj)
GetHashCode   Method        int GetHashCode()
GetType       Method        type GetType()
TcrptProprtp  Method        string ToString()
Age           NoteProperty  int Age==32
City          NoteProperty  string City=Amritsar
Job           NoteProperty  string Job=Lab Attendant
Name          NoteProperty  string Name=Lovejeet Singh
```

Object vs Text

Feature	Text	Object
Structure	Unstructured	Structured
Properties	No	Yes
Methods	No	Yes
Pipeline	Text only	Object (Rich data)
Easy to Work	Difficult	Easy & Powerful

PSObject Example

```
$obj = New-Object PSObject
$obj | Add-Member NoteProperty Name "Jeet"
$obj | Add-Member NoteProperty Age 32
$obj
$obj | Get-Member
```

PSObject is the base object in PowerShell.

Real World Example

```
# Get running services (objects)
Get-Service | Select-Object -First 2
  | Format-List Name, Status, StartType
```

Services are returned as objects with properties like Name, Status etc.

Pro Tips

- Use Get-Member to explore new objects.
- Use Select-Object to pick specific properties.
- Use Add-Member to extend objects.
- Objects make automation easy and powerful.
- Always prefer objects over plain text.

Common Operators with Objects

Operator	Example	Description
.	$obj.Name	Access property
|	$obj | Get-Member	Send object to command
()	($obj).Name	Group for precedence
Add-Member	Add-Member ...	Add new property/method

Object Workflow

Get Object (From Cmdlet) → Explore Members (Get-Member) → Access Properties (.Property) → Use Methods (Actions) → Pass in Pipeline (Next Cmdlet)

Note

- Objects are case-insensitive.
- Property names may vary by object type.
- Not all objects have methods.
- Use help(Get-Member) for more info.

Objects make PowerShell Powerful! Object • Properties • Methods • Get-Member • PSObject • Type • Value • Pipeline Automate Everything with Objects!

Pipeline | The Power of Flow

Pass Objects Between Commands

One Command. One Object. Smooth Processing.

Command 1	|	Command 2	Processing	Output	Objects	Pipeline
Generates Objects		Receives Objects as Input	Works on Objects (Processing)	Returns Objects (Output)	Objects Flow Through	The Pipeline (|) Connects Them

What is Pipeline?

The pipeline (|) passes the output (objects) of one command as input to the next command.

It works with objects, not just text!

Pipeline Concept

Input → Command 1 → (|) → Command 2 → Output

Objects flow from left to right through the pipeline.
Each command processes the objects and passes them forward.

Key Points

- Pipeline works with Objects, not plain text.
- Output of one command becomes Input of the next.
- Makes commands powerful and easy to combine.
- Each command in pipeline does one small task.
- Results are clean, structured and easy to use.

Benefits

- Simplifies complex tasks.
- Increases productivity.
- Reduces the need for temporary variables.
- Produces clean and meaningful output.

Remember

- Objects flow left to right.
- Use | to connect commands.
- Use Select-Object to shape output.
- Use Format-* cmdlets to format output.

1. Basic Pipeline Example

```
Get-Process | Where-Object { $_.CPU -gt 100 } | Sort-Object CPU -Descending
# Get running processes
# Filter processes where CPU > 100
# Sort the filtered objects by CPU (High to Low)
```

Get-Process (|) Where-Object (|) Sort-Object → Output

Get-Process	Where-Object	Sort-Object	Output
Command 1 (Generates Objects)	Command 2 (Filters Objects)	Command 3 (Sorts Objects)	Final Objects (Output)

2. See the Pipeline in Action

```
Get-Service | Where-Object { $_.Status -eq "Running" } | Select-Object Name, Status
# Get all services
# Only keep services that are Running
# Select and show Name and Status
```

Name	Status
AudioEndpointBuilder	Running
Dnscache	Running
PlugPlay	Running
Spooler	Running
...	...

Pipeline Flow

Objects Created (From Command 1)
↓
Objects Filtered (From Command 2)
↓
Objects Processed (From Command 3)
↓
Final Output (Objects Ready)

3. Pipeline with Select-Object

```
Get-Process | Select-Object Name, Id, CPU, PM
# Get processes
# Select only required properties
```

Name	Id	CPU	PM(K)
chrome	1234	125.5	98,304
Code	5678	45.2	55,120
explorer	2340	12.1	43,200
...	...	...	...

4. Pipeline with Measure-Object

```
Get-ChildItem | Measure-Object
# Get files and folders
# Count the total objects
```

```
Count    : 152
Average  :
Sum      :
Maximum  :
Minimum  :
Property :
```

5. Pipeline with ForEach-Object

```
Get-Process | ForEach-Object { $_.Name }
# Get processes
# For each object, show only Name
```

```
System
Registry
smss
csrss
wininit
...
```

Common Pipeline Commands

Where-Object	Filter objects
Select-Object	Select properties
Sort-Object	Sort objects
Measure-Object	Count / Measure
ForEach-Object	Perform action on each
Group-Object	Group objects
Out-File	Save output to file

Real World Example

Find the top 5 processes by memory usage:

```
Get-Process | Sort-Object PM -Descending |
Select-Object -First 5 Name, PM
```

Pipeline vs Traditional (Without Pipeline)

Without Pipeline (Old Way)

```
$procs = Get-Process
$filtered = $procs | Where-Object { $_.CPU -gt 100 }
$sorted = $filtered | Sort-Object CPU -Descending
$sorted
```

With Pipeline (Better Way)

```
Get-Process | Where-Object { $_.CPU -gt 100 } |
Sort-Object CPU -Descending
```

Pipeline Workflow

Command 1 → (|) → Command 2 → (|) → Command 3 → Output

Command 1	Command 2	Command 3	Output
(Create Objects)	(Filter Objects)	(Process Objects)	(Final Result)

Pro Tips

- Always think in objects.
- Use pipeline to chain small powerful commands.
- Use Select-Object to limit output.
- Use Format-Table or Format-List for viewing.
- Pipeline makes you a PowerShell Pro!

Pipeline = Power + Simplicity + Productivity ⚡ Command 1 | Command 2 | Command 3 | ... → Powerful Output Work Smarter with PowerShell Pipeline! 🚀

Where-Object

Filter Objects Like a Pro!

Powerful Filtering with Conditions

Where-Object	Where	$_	Property	Condition	Equal	Greater Than
Filters objects from the pipeline	Alias of Where-Object	Represents the current object	Object property used for filter	The rule to match objects	-eq Equal to	-gt Greater than

What is Where-Object?

Where-Object cmdlet filters objects based on a condition. It works on the pipeline and returns only the objects that match the condition.

Syntax

```
Where-Object Condition
# or using script block
Where-Object { $_.Property -eq "Value" }
```

Key Points

- Filters objects based on a condition.
- Uses $_ to represent the current object.
- Supports comparison operators like -eq, -gt, -lt, etc.
- Can use multiple conditions with -and, -or.
- Works with strings, numbers, dates, and more.

Comparison Operators

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-ge	Greater than or equal to
-le	Less than or equal to
-like	Like (pattern match)
-match	Regular expression match
-in	In collection
-notin	Not in collection

1. Basic Filtering Examples

Get processes with CPU greater than 100

```
Get-Process | Where-Object { $_.CPU -gt 100 }
```

Name	Id	CPU	PM(K)
chrome	1234	125.5	98,304
Code	5678	115.2	75,120
explorer	2340	102.8	43,200
...	...	...	...

Get services that are Running

```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

Name	Status
Spooler	Running
Dnscache	Running
AudioEndpointBuilder	Running
WinRM	Running
...	

Get files larger than 10MB

```
Get-ChildItem | Where-Object { $_.Length -gt 10MB }
```

Name	Length	LastWriteTime
video.mp4	52.3 MB	5/20/2024 10:15 AM
backup.zip	25.1 MB	5/18/2024 09:30 AM
data.iso	12.6 MB	5/19/2024 04:45 PM
...		

Quick Reference

Where-Object	Filter objects
$_	Current object
$_.Property	Access property
-eq	Equal
-ne	Not equal
-gt	Greater than
-lt	Less than
-ge	Greater or equal
-le	Less or equal
-like	Pattern match
-match	Regex match

2. Filter with String Matching

```
Get-Process | Where-Object { $_.ProcessName -like "*chrome*" }
```

Name	Id	CPU
chrome	1234	125.5
chrome.exe	3456	85.2
chromedriver	7890	12.3
...	...	...

3. Multiple Conditions (ANB)

```
Get-Process | Where-Object { $_.CPU -gt 50 -and $_.PM -gt 50000 }
```

Name	Id	CPU	PM(K)
chrome	1234	125.5	98,304
Code	5678	65.4	55,120
...	...	...	...

4. Multiple Conditions (OR)

```
Get-Process | Where-Object { $_.CPU -gt 100 -or $_.PM -gt 80000 }
```

Name	Id	CPU	PM(K)
chrome	1234	125.5	98,304
Teams	2345	45.2	82,112
...	...	...	...

Pro Tip

- Use Where-Object early in the pipeline to reduce data.
- Use -like for wildcards (*, ?).
- Combine conditions with -and / -or.
- Use parentheses () for complex logic.

Note

- Where-Object returns objects that match the condition only.
- $_ is an alias for $PSItem.
- Property names are case-insensitive.

5. Advanced Usage with Script Block

```
Get-Service | Where-Object { $_.StartType -eq "Automatic" -and $_.Status -eq "Running" }
```

Name	Status	StartType
Dnscache	Running	Automatic
W32Time	Running	Automatic
WinRM	Running	Automatic
...		

6. Filter by Date

```
Get-ChildItem | Where-Object { $_.LastWriteTime -gt (Get-Date).AddGays(-7) }
```

Name	LastWriteTime
report.docx	5/20/2024 9:10 AM
notes.txt	5/19/2024 3:22 PM
image.png	5/18/2024 11:05 AM
...	...

7. Using Where Alias

```
Get-Process | Where { $_.CPU -gt 100 }
```

Where is an alias of Where-Object. Both do the same job!

Pipeline Workflow

Command 1 (Generate Objects) → Where-Object (Filter Objects) → Command 2 (Next Step) → Output (Filtered Objects)

Where-Object = Filter + Focus + Power! 🚀 Command 1 | Where-Object { Condition } | Command 2 ... Filter Smart. Work Smart. PowerShell! 💪

Select-Object

Select & Shape Your Data Like a Pro!

Pick Properties, Filter Objects, Get Exactly What You Need.

Select Data	Properties	Objects	-Property	-First	-Last	-Unique
Choose specific data from objects	Pick one or more properties	Work with object collections	Select specific properties	Get first N objects	Get last N objects	Remove duplicates (Unique values)

What is Select-Object?

Select-Object cmdlet is used to select properties of objects or to limit the number of objects in the output. It helps in formatting and viewing data the way you want.

Syntax

```
Select-Object [ -Property <PropertyName[]> ]
[ -First <Number> ] [ -Last <Number> ]
[ -Skip <Number> ] [ -Unique ]
```

Key Points

- Use -Property to select specific properties.
- Use -First to get first N objects.
- Use -Last to get last N objects.
- Use -Skip to skip first N objects.
- Use -Unique to remove duplicate objects or values.
- Combine parameters to shape output.

Common Parameters

Parameter	Meaning
-Property	Selects one or more properties
-First	Returns first N objects
-Last	Returns last N objects
-Skip	Skips first N objects
-Unique	Returns unique objects or values
-ExcludeProperty	Excludes specific properties
-ExpandProperty	Expands property value
-Index	Selects item at specific index

Pro Tip

- Use * to select all properties.
- Use calculated properties with hashtable.
- Pipe output to Export-Csv, Format-Table, etc.
- Combine -Property with -First / -Last for best control.

1. Select Specific Properties

```
Get-Process | Select-Object -Property Name, Id, CPU, PM
```

Name	Id	CPU	PM(K)
chrome	1234	125.5	98,304
Code	5678	65.4	55,120
explorer	2340	12.1	43,200
...	...	...	...

2. Select All Properties (Default)

```
Get-Service | Select-Object * -First 3
```

Status	Name	DisplayName	StartType
Running	Spooler	Print Spooler	Automatic
Running	Dnscache	DNS Client	Automatic
Running	W32Time	Windows Time	Manual

3. Select First N Objects

```
Get-Process | Select-Object -First 5
```

Name	Id	CPU
System	4	0.1
smss	572	0.0
csrss	744	0.1
wininit	848	0.0
services	960	1.2

4. Select Last N Objects

```
Get-Process | Select-Object -Last 5
```

Name	Id	CPU
notepad	8420	0.5
Code	9212	3.2
chrome	10220	5.6
PowerShell	11040	1.8
devenv	11500	2.4

5. Skip First N Objects

```
Get-Process | Select-Object -Skip 5 -First 5
```

Name	Id	CPU
services	960	1.2
lsass	1012	0.7
svchost	1116	1.0
svchost	1220	2.1
dwm	1324	0.4

6. Select Unique Values

```
Get-Process | Select-Object -Property ProcessName -Unique
```

ProcessName
System
smss
csrss
wininit
services
...

7. Exclude Properties

```
Get-Process | Select-Object -ExcludeProperty Handles, VM
```

Name	Id	CPU	PM(K)
chrome	1234	125.5	98,304
Code	5678	65.4	55,120
explorer	2340	12.1	43,200
...	...	...	...

8. Expand Property (for collections)

```
Get-Process | Select-Object -ExpandProperty Modules -First 3
```

ModuleName
ntdll.dll
kernel32.dll
KERNELBASE.dll
...

9. Select by Index

```
(Get-Process)(0) | Select-Object *
```

Property	Value
Name	System
Id	4
CPU	0.1
PM(K)	2,100
Handles	210
...	...

10. Calculated Properties Example

```
Get-Process | Select-Object Name,
@{Name="MemoryMB";Expression={[math]::Round($_.PM/1MB,2)}}
```

Name	MemoryMB
chrome	96.00
Code	53.83
explorer	42.19
...	...

Pipeline Workflow

Command (Generate Objects) → Select-Object (Choose Data) → Shaped Objects (Selected Data) → Output (Display / Export)

Note

- Select-Object does not change the original objects, it only controls the output.
- Property names are not case-sensitive.
- Use Get-Member to see available properties.
- Use -Unique carafully; it compares entire object or selected property.

Select-Object = Control Your Output, Show What Matters! 🚀 | -Property • -First • -Last • -Skip • -Unique • -ExcludeProperty • -ExpandProperty Smart Output. Better PowerShell! 💪

Sort-Object

Sort Objects & Data Like a Pro!

Order your data using properties, ascending or descending.

Sort	-Property	Ascending (Default)	-Descending	Objects	Data
Arranges objects in a specific order	Sort by one or more properties	Small to Large A to Z	Large to Small Z to A	Works with all PowerShell objects	Sort text, numbers, dates & more

What is Sort-Object?

Sort-Object cmdlet sorts the objects by the values of the specified property. By default, it sorts in ascending order. Use -Descending for reverse order.

Syntax

```
Sort-Object -Property <PropertyName> [-Descending]
Sort-Object [-Property <Proprtey1, Property2>]
```

Key Points

- Sorts objects by the values of a property.
- Default order is Ascending (A to Z / 0 to 9).
- Use -Descending for reverse order.
- Supports multiple properties for advanced sorting.
- Works with numbers, strings, dates and more.

Common Parameters

Parameter	Dsscription
-Property	Specifies the property to sort by
-Descending	Sorts in descending order
-Unique	Returns unique objects
-CaseSensitive	Performs case-sensitive sort
-Top	Returns top N objects after sort

Pro Tip

- Combine with Get-Process, Get-Service, Get-Childitem.
- Use multiple properties for precise sorting.
- Pipe output to Select-Object to show specific columns.
- Use -Descending to quickly find highest/most recent.

Remember

- Default order is Ascending.
- Numbers sort numerically, not as text.
- Dates sort by oldest to newest (ascending).
- Strings sort alphabetically.

1. Basic Sort (Ascending - Default)

```
Get-Process | Sort-Object CPU
```

Name	Id	CPU
System	4	0.05
smss	572	0.12
wininit	848	0.23
svchost	1116	1.05
chrome	1234	12.55
...	...	...

Sort by CPU in ascending order (low to high).

2. Descending Sort

```
Get-Process | Sort-Object CPU -Descending
```

Name	Id	CPU
chrome	1234	12.55
explorer	2340	2.32
svchost	1116	1.05
wininit	848	0.23
System	4	0.05
...	...	...

Sort by CPU in descending order (high to low).

3. Sort by Name (A to Z)

```
Get-Process | Sort-Object Name
```

Name	Id	CPU
chrome	1234	12.55
Code	5678	1.21
explorer	2340	2.32
powershell	6780	0.65
svchost	1116	1.05
...	...	...

Sort by Name in alphabetical order (A to Z).

4. Sort by Name (Z to A)

```
Get-Process | Sort-Object Name -Descending
```

Name	Id	CPU
svchost	1116	1.05
powershell	6780	0.65
explorer	2340	2.32
Code	5678	1.21
chrome	1234	12.55
...	...	...

Sort by Name in reverse alphabetical order.

5. Multiple Properties Sort

```
Get-Process | Sort-Object CPU, Name
```

Name	Id	CPU
System	4	0.05
smss	572	0.12
wininit	848	0.23
powershell	6780	0.65
Code	5678	1.21
...	...	...

First by CPU, then by Name.

6. Sort Services by Status

```
Get-Service | Sort-Object Status, Name
```

Status	Name	DisplayName
Running	AudioEndpointBuilder	Windows Audio...
Running	Dnscache	DNS Client
Running	Spooler	Print Spooler
Stopped	wuauserv	Windows Update
Stopped	BITS	Background Int...
...	...	...

Running services first, then Stopped, sorted by Name.

7. Sort Files by LastWriteTime

```
Get-ChildItem | Sort-Object LastWriteTime
```

Name	LastWriteTime	Length
report.docx	5/20/2024 10:15 AM	52 KB
data.xlsx	5/19/2024 09:30 AM	25 KB
notes.txt	5/18/2024 04:45 PM	12 KB
image.png	5/18/2024 11:05 AM	3 MB
backup.zip	5/17/2024 08:20 PM	2 MB
...	...	...

Oldest files first (ascending by date).

8. Latest Files First (Descending)

```
Get-ChildItem | Sort-Object LastWriteTime -Descending
```

Name	LastWriteTime	Length
seport.docx	5/20/2024 10:15 AM	52 KB
Sora.xlsx	5/19/2024 09:30 AM	25 KB
image.png	5/18/2024 11:05 AM	3 MB
notes.txt	5/18/2024 04:45 PM	12 KB
backup.zip	5/17/2024 08:20 PM	2 MB
...	...	...

Newest files first (descending by date).

9. Unique Sorted Values

```
Get-Process | Sort-Object Name -Unique
```

Returns unique process names sorted alphabetically.

Example Output:

Code, chrome, conhost, explorer, powershell, svchost, ...

...

Sorting Order Summary

Ascending (Default)	Descending (-Descending)
A to Z	Z to A
0 to 9	9 to 0
Oldest to Newest	Newest to Oldest
Low to High	High to Low

Pipeline Workflow

Get Data (Objects) → Sort-Object (Sort Data) → Select-Object (Choose Columns) → Export / Display (Output)

Common Use Cases

- Sort processes by CPU/Memory usage.
- Sort services by status and name.
- Sort files by date, size or name.
- Find top or latest data easily.
- Prepare clean, ordered data for reports.

Note

- Sort-Object does not change source data.
- It only sorts the output.
- Combine with Where-Object, Select-Object
- for powerful results.

Sort-Object = Order + Clarity + Control! 🔥 Get-Data | Sort-Object -Property <Name> [-Descending] | Select-Object ... Sort Smart. Work Fast. PowerShell! 💪

ForEach-Object

Automate Repetitive Tasks Like a Pro! ⚡

Loop through pipeline objects and perform actions on each item.

ForEach	$_	Process	Loop	Objects	Pipeline	Automation
Loop through each object	Represents current object	Perform action on each object	Repeats for every object	Works with any pipeline objects	Accepts input from pipeline	Powerful for automation

What is ForEach-Object?

ForEach-Object cmdlet loops through each object in the pipeline and runs a script block ({ }) for every object.
It is the PowerShell way of "for each loop".

Syntax

```
Get-Command | ForEach-Object { $_.Name }
# or
Get-Command | ForEach { $_.Name }
```

Alias: % (percent) is also supported.
Example: `Get-Process | % { $_.Name }`

Key Points

- Processes one object at a time from the pipeline.
- $_ represents the current object.
- Use script block { } to define the action.
- Works with any type of objects.
- Great for automation and bulk operations.

Common Parameters

Parameter	Description
-Process	Script block to run for each object
-Begin	Script block to run once before loop
-End	Script block to run once after loop
-Parallel	Run script block in parallel (PowerShell 7+)
-ThrottleLimit	Limit number oti parallel runspaces

Pro Tip

- Use $_ instead of full property path to save time.
- Combine with Where-Object, Select-Object, etc.
- Use -Parallel for faster processing in PowerShell 7+.
- Keep script block lightweight for performance.

1. Basic Examples

List all process names

```
Get-Process | ForEach-Object { $_.Name }
```

Output (Process Names)

```
System
smss.exe
csrss.exe
wininit.exe
services.exe
...
```

Show process name and ID

```
Get-Process | ForEach-Object {
  "$($_.Id) - $($_.Name)"
}
```

Output

```
4 - System
572 - smss.exe
744 - csrss.exe
848 - wininit.exe
...
```

Display service status

```
Get-Service | ForEach-Object {
  "$($_.Status) - $($_.Name)"
}
```

Output

```
Running - Spooler
Running - Dnscache
Running - AudioEndpointBuilder
Stopped - wuauserv
...
```

Get file sizes in MB

```
Get-ChildItem -File | ForEach-Object {
  "$($_.Name) - $([math]::Round($_.Length/1MB,2)) MB"
}
```

Output

```
report.docx - 1.25 MB
notes.txt - 0.12 MB
image.png - 2.45 MB
backup.zip - 12.30 MB
...
```

Convert text to uppercase

```
"hello","powerShell","world" |
ForEach-Object { $_.ToUpper() }
```

Output

```
HELLO
POWERSHELL
WORLD
```

2. Advanced Usage

Use with Where-Object (Filter & Process)

```
Get-Process | Where-Object { $_.CPU -gt 10 } |
ForEach-Object { "High CPU: $($_.Name) ($($_.CPU)%)" }
```

Example Output

```
High CPU: chrome (25.5%)
High CPU: code (18.2%)
High CPU: explorer (12.8%)
```

With Select-Object (Custom Output)

```
Get-Process | Select-Object Name,CPU,PM |
ForEach-Object { "(0,-20} (1,5) (2,6) MB" -f
$_.Name, $_.CPU, [math]::Round($_.PM / 1MB,2) }
```

Example Output

```
chrome      25.5   98.30 MB
code        18.2   75.12 MB
explorer    12.8   43.21 MB
```

Begin / Process / End Blocks

```
Get-Process |
ForEach-Object -Begin { $count = 0 }
  -Process { $count++ }
-End { "Total Processes: $count" }
```

Output

```
Total Processes: 142
```

Using -Parallel (PowerShell 7+)

```
Get-ChildItem -File |
ForEach-Object -Parallel {
   [math]::Round($_.Length / 1MB,2)
} -ThrottleLimit 5
```

Output (File Sizes in MB)

```
1.25
0.12
2.45
12.30
...
```

With Alias % (percent)

```
Get-Service | % { "$($_.Name) - $($_.Status)" }
```

Output

```
Spooler - Running
Dnscache - Rnnning
BITS - Stopped
wuauserv - Stopped
...
```

3. Real World Use Cases

- Find & process large log files
- Bulk rename or modify files
- Process database query results
- Monitor & report system data
- Automate reports & exports

4. Pipeline Workflow

Get Data (Pipeline Input) → Filter Data (Optional) → ForEach-Object (Process Each Item) → Perform Action (Script Block) → Output Result (Display / Save)

Note

- ForEach-Object is memory efficient.
- Avoid heavy operations in script block.
- Use -Parallel carefully to avoid high CPU/Memory usage.
- Best practice: Keep script blocks small and focused.

For Each Object, Do More! 🚀 | Pipeline + ForEach-Object = Automation Power 💪 Work Less. Automate More. PowerShell! 😎

Measure-Object

Measure Everything Like a Pro!

Get statistics and measurements from your data in seconds.

-Property	-Count	-Sum	-Average	-Minimum	-Maximum	Statistics
Measure specific properties	Count the number of objects	Calculate the sum of values	Calculate the average	Find the minimum value	Find the maximum value	Powerful statistics in one command

What is Measure-Object?

Measure-Object cmdlet calculates statistical information about objects in the pipeline. It can count, sum, average, find min/max values and more.

Syntax

```
Get-Command | Measure-Object
Get-Process | Measure-Object -Property CPU
Get-ChildItem | Measure-Object -Property Length -Sum
Get-Process | Measure-Object -Property CPU -Average
```

Key Points

- Works with any pipeline objects.
- Provides statistical summary of numeric properties.
- Use -Property to measure specific properties.
- Can calculate Count, Sum, Average, Min, Max.
- Ignores non-numeric values automatically.
- Perfect for reports and data analysis.

Common Parameters

Parameter	Description
-Property	Specify property to measure
-Average	Calculate average value
-Sum	Calculate sum of values
-Minimum	Find minimum value
-Maximum	Find maximum value
-Count	Count number of objects
-StandardDeviation	Calculate standard deviation

1. Basic Usage Examples

Count total number of objects

```
Get-Process | Measure-Object
```

Output

```
Count     : 146
Average   :
Sum       :
Maximum   :
Minimum   :
Property  :
```

Count lines in a file

```
Get-Content .\log.txt | Measure-Object
```

Output

```
Count     : 1250
Average   :
Sum       :
Maximum   :
Minimum   :
Property  :
```

Calculate total size of files (Sum)

```
Get-ChildItem -File |
Measure-Object -Property Length -Sum
```

Output (in Bytes)

```
Count     :
Average   :
Sum       : 1256738472
Maximum   :
Minimum   :
Property  : Length
```

Average CPU usage of processes

```
Get-Process |
Measure-Object -Property CPU -Average
```

Output

```
Count     :
Average   : 1.87
Sum       :
Maximum   :
Minimum   :
Property  : CPU
```

Min & Max Working Set (Memory)

```
Get-Process |
Measure-Object -Property WS -Minimum,Maximum
```

Output

```
Count     : .
Average   : .
Sum       : .
Maximum   : 45219840
Minimum   : 8187904
Property  : WS
```

2. Detailed Statistics (All in One)

```
Get-Process | Measure-Object -Property CPU -Average -Sum -Minimum -Maximum
```

Output

```
Count     : 146
Average   : 1.87
Sum       : 272.35
Maximum   : 25.40
Minimum   : 0.00
Property  : CPU
```

3. Measure Multiple Properties

```
Get-Process | Measure-Object -Property CPU,WS -Average -Minimum -Maximum
```

Output

Property	Count	Average	Minimum	Maximum	Sum
CPU	146	1.87	0.00	25.40	272.35
WS	146	21504832	8187904	45219840	3138713216

4. Text Statistics (String Length)

```
Get-ChildItem -File | Measure-Object -Property Name -Average
```

Output

```
Count     : 45
Average   : 19.62
Sum       :
Maximum   : 56
Minimum   : 5
Property  : Name
```

5. With Where-Object (Filtered Data)

```
Get-Process | Where-Object {$_.CPU -gt 5} |
Measure-Object -Property CPU -Average -Maximum
```

Output

```
Count     : 12
Average   : 12.45
Sum       : 149.37
Maximum   : 25.40
Minimum   : 5.01
Property  : CPU
```

Pro Tip

- Use -Property for specific columns.
- Combine with Sitect-Object and Where-Object.
- Great for performance monitoring and reports.
- Use -Average, -Sum, -Min, -Max as needed.
- Pipe output to Export-Csv for documentation.

6. File Size Statistics (Human Readable)

```
Get-ChildItem -File |
Measure-Object -Property Length -Sum |
Select-Object @{n='TotalSize(MB)';e={[math]::Round($_.Sum/1MB,2)}}
```

Output

```
TotalSise(MB)

...    ''
```

7. Count Specific Items

```
Get-ChildItem -File -Recurse | [illegible]
[illegible] {$_.Extension -eq '.log'} |
Measure-Object
```

Output

```
Count     : 78
Average   :
Sum       :
Maximum   :
Minimum   :
Property  :
```

Understanding Output

Field	Meaning
Count	Total number of objects
Average	Average of numeric values
Sum	Sum of all numeric values
Maximum	Highest numeric value
Minimum	Lowest numeric value
Property	Property that was measured

Pipeline Workflow

Get Data (Pipeline Input) → Filter / Select (Optional) → Measure-Object (Calculate Stats) → Display Results (Output)

Note

- Measure-Object works only on numeric values for calculations.
- Non-numeric values are ignored.
- Use -Property to specify exact column.
- Great for reports, monitoring & analytics.

Measure Data. Get Insights. Make Better Decisions! | Measure-Object = Statistics in One Command! | -Property • -Count • -Sum • -Average • -Minimum • -Maximum | PowerShell Makes It Easy! ⚡

Format Commands

Display Data the Way You Want! 😎

Control how objects are displayed in PowerShell.

Format-Table	Format-List	Format-Wide	Format-Custom	Display	Output
Display data in table format	Display data in list format	Display data in wide columns	Custom view of object properties	Control the output view	Beautiful & Readable Output

What are Format Commands?

Format commands control how objects are shown in the console. They do not change the data, only the display.
Use them at the end of a pipeline to format the output.

Syntax

```
Get-Command  | Format-Table
Get-Process  | Format-List
Get-Service  | Format-Wide
Get-ChildItem | Format-Custom
```

Key Points

- Format commands only change the display.
- They do not affect or modify the objects.
- Use them at the end of the pipeline.
- Choose the format based on data and need.
- Makes output clean, easy to read & useful.

Common Parameters

Parameter	Description
-AutoSize	Adjust columns automatically
-Property	Select specific properties
-Wrap	Wrap long lines
-HideTableHeaders	Hide headers in table
-GroupBy	Group data before formatting

Pro Tip

- Use -AutoSize for better table view.
- Use -Property to show only needed data.
- Format-List is best for detailed information.
- Format-Wide is great for short lists.
- Format-Custom for advanced / custom views.

1. Format Commands Overview & Examples

Format-Table

Display data in table (default) format.

```
Get-Process | Select-Object Name, Id, CPU |
Format-Table -AutoSize
```

Output (Table)

Name	Id	CPU
chrome	1234	125.5
Code	5678	65.4
explorer	2340	12.1
powershell	6780	45.2
...	...	...

Best for:

- General data in rows & columns
- Easy comparison of values

Format-List

Display data in list (vertical) format.

```
Get-Process -Id 1234 | Format-List *
```

Output (List)

```
Name            : chrome
Id              : 1234
CPU             : 125.5
Handles         : 2101
WorkingSet(MB)  : 230.45
Path            : C:\Program Files\Google\Chrome\...
Company         : Google LLC
...
```

Best for:

- Detailed information about one/few objects
- Readable key : value format

Format-Wide

Display data in wide columns.

```
Get-Service | Select-Object -First 6 Name, Status |
Format-Wide -Column 3
```

Output (Wide)

Name	Status	Name	Status	Name	Status
Spooler	Running	Dnscache	Running	W32Time	Running
AudioSrv	Running	winrm	Running	BITS	Stopped

Best for:

- Short lists in wide view
- Save vertical space

Format-Custom

Custom display with selected properties.

```
Get-Process -Id 1234 |
Format-Custom -Property Name, Id, CPU, Handles
```

Output (Custom View)

```
Process Info (Custom View)
--------------------------
Name      Id     CPU     Handles
chrome    1234   125.5   2101
```

Best for:

- Custom layouts
- Advanced & clean dashboard style output

2. More Examples

Table with Selected Properties

```
Get-Process | Select-Object Name, CPU, PM |
Format-Table -AutoSize
```

Output

Name	CPU	PM(K)
chrome	125.5	98,304
Code	65.4	55,120
explorer	12.1	43,200

List with Specific Properties

```
Get-Service Spooler |
Format-List Name, Status, StartType
```

Output

```
Name       : Spooler
Status     : Running
StartType  : Automatic
```

Wide with Auto Size

```
Get-Process | Select-Object -First 4 Name, Id, CPU |
Format-Wide -AutoSize
```

Output

Name	Id	CPU	Name	Id	CPU
chrome	1234	125.5	Code	5678	65.4
explorer	2340	12.1	powershell	6780	45.2

Custom View (Advanced)

```
Get-Service | Where-Object {$_.Status -eq 'Running'} |
Select-Object Name, Status, StartType |
Format-Custom
```

Output (Custom)

```
Service Dashboard
------------------------------------------
Name        Status     StartType
Spooler     Running    Automatic
Dnscache    Running    Automatic
...
```

Quick Summary

Format-Table	Table view
Format-List	List (key:value)
Format-Wide	Wide columns
Format-Custom	Custom layout

When to Use What?

- Need comparison? → Format-Table
- Need details? → Format-List
- Short list, save space? → Format-Wide
- Custom clean view? → Format-Custom

⚠ Note

- Format commands only affect display.
- Data is not changed or filtered.
- Place format command at the end of the pipeline.
- Combine with Select-Object for best results.

Format Your Output. See Better. Work Smarter! 💡 | Format-Table | Format-List | Format-Wide | Format-Custom | Clear Output. Better Understanding. PowerShell! ⚡

Export & Import

Move Your Data Anywhere!

Export Data to Files or Import Data from Files Easily.

Export-Csv	Import-Csv	Export-Clixml	Import-Clixml	CSV	XML	Data	Objects
Export data to CSV file	Import data from CSV file	Export objects to CLIXML file	Import objects from CLIXML file	Comma Separated Values	Extensible Markup Language	Store & Transfer Data	Preserve Object Properties

Why Export / Import?

- ✔ Save data to use later
- ✔ Share data with others
- ✔ Backup important information
- ✔ Move objects between systems
- ✔ Work with data in external tools

Syntax

```
Export-Csv     -Path <file.csv> [-NoTypeInformation]
Import-Csv     -Path <file.csv>
Export-Clixml  -Path <file.xml>
Export-Clixml  -Path <file.xml>
```

Key Points

- Export-Csv exports objects to CSV (table) format.
- Import-Csv imports CSV data as custom objects.
- Export-Clixml exports objects to XML with full object information.
- Import-Clixml imports XML and restores objects with all properties.
- Use -NoTypeInformation with Export-Csv to remove type header.

Common Parameters

Parameter	Description
-Path	File path for import or export
-NoTypeInformation	(Export-Csv) Removes type header
-Delimiter	(Export/Import-Csv) Set delimiter (default ,)
-Encoding	(Export/Import) Set file encoding
-Force	Overwrite file if it already exists

Pro Tip

- Use CSV for simple tabular data (Excel friendly).
- Use CLIXML for PowerShell objects (keeps data types, properties, methods).
- CLIXML files are not human readable, CSV is.
- Combine with pipeline for powerful automation!

1 Export-Csv

Export objects to CSV file.

```
Get-Process | Select-Object Name, Id, CPU |
Export-Csv -Path .\processes.csv -NoTypeInformation
```

Output File: processes.csv

```
Name,Id,CPU
chrome,1234,125.5
Code,5678,65.4
explorer,2340,12.1
powershell,6780,45.2
...
```

Use when:

- Need data in table format
- Open in Excel or other tools
- Share simple data

2 Import-Csv

Import data from CSV file as objects.

```
Import-Csv -Path .\processes.csv
```

Output (Objects)

Name	Id	CPU
chrome	1234	125.5
Code	5678	65.4
explorer	2340	12.1
powershell	6780	45.2
...		

Use when:

- Read CSV data back into PowerShell
- Process imported data
- Filter, sort or analyze

3 Export-Clixml

Export objects to CLIXML file (full object data).

```
Get-Service |
Export-Clixml -Path .\services.xml
```

Output File: services.xml

```
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/
powershell/2004/04">
  <Obj RefId="0">
    <TN RefId="0">
      <T>System.ServiceProcess.ServiceController</T>
      <T>System.ComponentModel.Component</T>
    </TN>
    ...
  </Obj>
  ...
</Objs>
```

Use when:

- Need to save full objects (properties + methods)
- Preserve data types
- Restore exact objects later

4 Import-Clixml

Import objects from CLIXML file.

```
Import-Clixml -Path .\services.xml
```

Output (Objects)

Name	Status	DisplayName
Spooler	Running	Print Spooler
Dnscache	Running	DNS Client
wuauserv	Running	Windows Update
BITS	Stopped	Background Intelligent Transfer
...		

Use when:

- Restore full PowerShell objects
- Continue working with saved objects
- Objects behave exactly the same

Real World Examples

Export Process Data to CSV

```
Get-Process | Select-Object Name,Id,CPU |
Export-Csv -Path .\processes.csv -NoTypeInformation
```

Export Services to XML

```
Get-Service |
Export-Clixml -Path .\services.xml
```

Import & Filter

```
Import-Csv -Path .\processes.csv | Where-Object {
[double]$_.CPU -gt 50 } | Format-Table
```

Import & Use

```
$services = Import-Clixml -Path .\services.xml
$services | Where-Object { $_.Status -eq 'Running' }
| Select-Object Name,Status
```

File Comparison

Feature	CSV	CLIXML
Format	Text (Table)	XML (Object)
Human Readable	Yes	No
Preserves Data Types	No (All as Text)	Yes
Preserves Methods	No	Yes
Best For	Simple Data	PowerShell Objects
Open in Excel	Yes	No

Pipeline Workflow

Get Data (Objects) → Export (Save) → File (CSV / XML) → Import (Load) → Process Data (Objects)

Note

- CSV does not preserve data types (numbers become text).
- CLIXML files are PowerShell specific.
- Do not edit CLIXML files manually.
- Use correct path and file permissions.

Export Your Data. Import Your Success! | Export-Csv • Import-Csv • Export-Clixml • Import-Clixml | CSV • XML • Data • Objects | PowerShell Makes Data Easy!

JSON

Work with JSON Data Easily in PowerShell! 🚀

Convert, Parse and Work with Data in JSON Format.

ConvertTo-Json	ConvertFrom-Json	JSON	Object	API	Data	Convert	Parse
Convert Object to JSON	Convert JSON to Object	JavaScript Object Notation	PowerShell Objects	Used in REST APIs	Structured Data	Transform Data	Read & Analyze JSON

What is JSON?

JSON (JavaScript Object Notation) is a lightweight format for storing and exchanging data. It is easy for humans to read and easy for machines to parse.

Syntax

```
# Convert Object to JSON
$Object | ConvertTo-Json [Options]

# Convert JSON to Object
$Json | ConvertFrom-Json [Options]
```

Key Points

- ConvertTo-Json converts PowerShell objects to JSON.
- ConvertFrom-Json converts JSON text to objects.
- Use -Depth to include nested properties.
- JSON is widely used in Web APIs and Config files.
- Supports arrays, objects, numbers, strings, booleans.

Common Parameters

Parameter	Description
-Depth	Specifies the depth of nested objects
-Compress	Remove extra whitespace from JSON
-EnumsAsStrings	Convert enum values to strings
-AsHashtable	Return hashtable instead of PSCustomObject

Pro Tip

- Always use -Depth for complex objects.
- Use -Compress to get compact JSON.
- ConvertFrom-Json output is PSCustomObject.
- Great for API requests & configuration files!

1 ConvertTo-Json – Object to JSON

Convert PowerShell objects to JSON string.

```
$Data = Get-Process | Select-Object -First 2 Name, Id, CPU
$Data | ConvertTo-Json -Depth 3
```

Output (JSON)

```
[
  {
    "Name": "chrome",
    "Id" : 1234,
    "CPU": 45.2
  },
  {
    "Name": "powershell",
    "Id" : 6780,
    "CPU": 12.1
  }

]
```

Options

```
-Depth 3          # Include nested objects
-Compress         # Minify JSON output
-EnumsAsStrings # Enum as strings
```

Use when:

- Sending data to Web APIs
- Saving data to JSON files
- Sharing data between systems

2 ConvertFrom-Json – JSON to Object

Convert JSON string back to PowerShell objects.

```
$Json = Get-Content .\data.json -Raw
$Object = $Json | ConvertFrom-Json
```

Input (JSON)

```
{
  "Name": "Server01",
  "IP": "192.168.1.10",
  "Status": "Running",
  "Services": [
    "W3SVC",
    "WinRM",
    "BITS"
  ]
}
```

Output (Object)

Property	Value
Name	Server01
IP	192.168.1.10
Status	Running
Services	{W3SVC, WinRM, BITS}

Use when:

- Reading JSON from files or APIs
- Processing configuration or logs
- Converting API responses to objects

JSON Data Types

Type	Example (JSON)	Description
String	"PowerShell"	Text value
Number	123, 45.67	Numeric value
Boolean	true, false	True or False
Array	["a", "b", "c"]	Ordered list
Object	{"name": "A"}	Key-value pairs
Null	null	Empty value

JSON in Real World (API Example)

GET Request Example

```
$Response = Invoke-RestMethod -Uri "https://api.github.com/user"
$Response | ConvertTo-Json -Depth 3
```

Parsed Response (Object)

Property	Value
login	octocat
id	583231
name	The Octocat
public_repos	8
followers	3930
...	...

3 Practical Examples

1. Export Object to JSON File

```
Get-Service | Select-Object Name, Status |
ConvertTo-Json -Depth 3 |
Out-File .\services.json -Encoding UTF8
```

Result: services.json file created ✅

2. Read JSON File and Use Data

```
$Json = Get-Content .\config.json -Raw |
ConvertFrom-Json
$Json.AppName
```

Result: Use JSON data in scripts ✅

3. API Call and Convert Response

```
$Res = Invoke-RestMethod -Uri
"https://api.github.com/users/octocat"
$Res | ConvertTo-Json -Depth 3
```

Result: API data in JSON format ✅

4. Work with Nested JSON

```
$Json = Get-Content .\data.json -Raw |
ConvertFrom-Json
$Json.user.name
```

Result: Access nested properties ✅

Sample Object (PowerShell)

```
$Obj = [PSCustomObject]@{
    Name       = "Server01"
    IP         = "192.168.1.10"
    Status     = "Running"
    Services   = @("W3SVC","WinRM","BITS")
}
```

Converted to JSON

```
{
  "Name": "Server01",
  "IP":  "192.168.1.10",
  "Status": "Running",
  "Services": ["W3SVC","WinRM","BITS"]
}
```

Back to Object

```
$Json | ConvertFrom-Json
# Returns PSCustomObject

$Object.Name            # Server01
$Object.Services[0]   # W3SVC
```

File Workflow

Note

- JSON is case-sensitive.
- Numbers remain numbers (not text).
- Use -Depth for nested objects.
- Large data may need -Compress.
- Always validate JSON before use.

Convert. Parse. Automate. ⚡ | Object ⇄ JSON | ConvertTo-Json • ConvertFrom-Json | JSON Makes Data Easy to Share! 🌐

SYSTEM INFORMATION

Get System Details Easily in PowerShell! 🚀

Useful Commands to Get System, Windows & PowerShell Information.

System Details | Windows Info | PowerShell Version | Date & Time | Host Info | Computer Name | Information Commands

Why System Information?

- ✔ Know your system details
- ✔ Check PowerShell version
- ✔ Get current date and time
- ✔ Useful for scripting & troubleshooting
- ✔ Works on Local & Remote Systems

Syntax

```
Get-ComputerInfo
$PSVersionTable
Get-Date
Get-Host
hostname
```

Key Points

- Get-ComputerInfo gives complete system details.
- $PSVersionTable shows PowerShell version info.
- Get-Date shows current date & time.
- Get-Host shows information about the host.
- hostname shows the computer name.

Common Parameters

Parameter	Description
-ComputerName	Specify remote computer name
-Property	Select specific properties
-Format-List	Display output in list format
-Property *	Get all properties
-ErrorAction	Define error handling behavior

Pro Tip

- Use Get-ComputerInfo -Property * for all details.
- Use $PSVersionTable for quick version check.
- Get-Host helps to know host environment.
- Combine commands for better reporting.

1 Get-ComputerInfo

Get detailed information about the computer.

```
Get-ComputerInfo
```

Output (Sample)

```
WindowsProductName    : Windows 11 Pro
WindowsVersion        : 23H2
OsBuild               : 22631
CsName                : DESKTOP-7Q9K2V1
CsDNSHostName         : DESKTOP-7Q9K2V1
OsArchitecture        : 64-bit
OsLanguage            : en-IN
TotalPhysicalMemory   : 16 GB
WindowsDirectory      : C:\Windows
SystemType            : Desktop
...                   :
```

✅ Use -Property to get specific information.

2 $PSVersionTable

Get PowerShell version information.

```
$PSVersionTable
```

Output (Sample)

```
Name                        Value
----                        -----
PSVersion                   5.1.22621.2506
PSEdition                   Desktop
PSCompatibleVersions        {1.0, 2.0, 3.0, 4.0...}
BuildVersion                10.0.22621.2506
CLRVersion                  4.0.30319.42000
WSManStackVersion           3.0
PSRemotingProtocolVersion : 2.3
SerializationVersion      : 1.1.0.1
...
```

✅ Shows PowerShell version & environment info.

3 Get-Date

Get current date and time.

```
Get-Date
```

Output (Sample)

```
01 June 2026  11:45:30 AM
Monday, 01 June 2026 11:45:30
```

✅ Use -Format for custom date format.

4 Get-Host

Get information about the current host.

```
Get-Host
```

Output (Sample)

```
Name              : ConsoleHost
Version           : 5.1.22621.2506
InstanceId        : 3f6a9c2b-7e2d-4f37-
                    a47a-5e3d4f6c1b2a
UI                : System.Manegement.
                    Automation.Internal.
                    Host.InternalHostUser
CurrentCulture    : en-IN
CurrentUICulture  : en-IN
PrivateData       : Microsoft.PowerShell.
                    ConsoleHost+ConsoleColor
DebuggerEnabled   : True
IsRunspacePushed  : False
```

✅ Shows host type, version & environment.

5 hostname

Get the computer (hostname) name.

```
hostname
```

Output (Sample)

```
DESKTOP-7Q9K2V1
```

✅ Quick way to get computer name.

Combine for Complete Report

```
Get-ComputerInfo | Select WindowsProductName, WindowsVersion,
OsBulid, OsArchitecture
$PSVersionTable
Get-Date Ftable
Get-Host
hostname
```

Benefit:

Get all important system information in one place.

Sample Report Output

```
Computer Name       : DESKTOP-7Q9K2V1
Windows Edition     : Windows 11 Pro
Windows Version     : 23H2 (Build 22631)
Architecture        : 64-bit
PowerShell Version  : 5.1.22621.2506
Date & Time         : 01 June 2026 11:45:30 AM
Host                : ConsoleHost
```

Use Cases

- System inventory and reporting
- Troubleshooting and diagnostics
- Environment validation before scripts
- Log system details for audit

Quick Reference Table

Command	Purpose
Get-ComputerInfo	Get detailed system info
$PSVersionTable	Get PowerShell version info
Get-Date	Get current date and time
Get-Host	Get host environment info
hostname	Get computer name

⚠ Note

- Get-ComputerInfo may take time on remote systems.
- Some properties require admin privileges.
- Output may vary based on Windows version.
- Use these commands for scripts for automation.

System Info Workflow

Run Command → Collect Info → Process Data → Generate Report → Share / Save

⭐ Summary

These commands help you get system, Windows & PowerShell information quickly and easily.
Essential for admins, developers & IT professionals.

Know Your System. Stay Ahead! 💡 | Get-ComputerInfo • $PSVersionTable • Get-Date • Get-Host • hostname | PowerShell – Powerful. Simple. Efficient. 🚀

DISK INFORMATION

Manage and Get Disk, Volume & Storage Information Easily in PowerShell!

Disk	Volume	Partition	PSDrive	Storage	Usage	Manage	Info
Physical Disk Information	Logical Volume Details	Partition Table Information	PowerShell Drives	Storage & Space Analysis	Space Usage & Capacity	Manage Disks & Volumes	Detailed Information

Why Disk Information?

- ✔ Check disk health, size & type
- ✔ Monitor volumes and free space
- ✔ Manage partitions and drives
- ✔ Useful for system admin & troubleshooting
- ✔ Works on Local & Remote Systems

Syntax

```
Get-Disk
Get-Volume
Get-Partition
Get-PSDrive
```

Key Points

- Get-Disk shows physical disk details.
- Get-Volume shows logical volume information.
- Get-Partition shows partition table details.
- Get-PSDrive shows PowerShell drives.
- Helps in storage monitoring & management.

Common Parameters

Parameter	Description
-Name	Specify name or drive letter
-CimSession	Run command on remote system
-Verbose	Show detailed output
-ErrorAction	Define error handling behavior
-EA	Shortcut for ErrorAction

Pro Tip

- Use Get-Volume for user-friendly output.
- Get-PSDrive shows all PowerShell drives.
- Check free space with SizeRemaining property.
- Combine with Where-Object for filtering.

1 Get-Disk

Get physical disk information.

```
Get-Disk
```

Output (Sample)

Number	Friendly Name	Serial Number	Size	BusType
0	Samsung SSD 870	S5Y2NF0T123456	465.76 GB	SATA
1	ST1000DM010-2EP102	ZN1A1ABC	931.51 GB	SATA
2	Kingston DataTraveler 3.0	001CC0E0E51C4D113	14.91 GB	USB
	...			

Use when:

- You want to see all physical disks in the system.
- Check disk size, type, model and status.

2 Get-Volume

Get logical volume (drive) information.

```
Get-Volume
```

Output (Sample)

DriveLetter	FileSystem	Label	Size	SizeRemaining
C	NTFS	Windows	237.37 GB	82.45 GB
D	NTFS	Data	931.50 GB	326.12 GB
E	NTFS	New Volume	465.75 GB	123.80 GB
F	FAT32	USB Drive	14.90 GB	7.20 GB

Use when:

- You want logical drive and free space details.
- Best for checking storage usage.

3 Get-Partition

Get partition information on disks.

```
Get-Partition
```

Output (Sample)

DiskNumber	PartitionNumber	Type	Size	GptType
0	1	System	100 MB	EFI System
0	2	Reserved	16 MB	Microsoft Reserved
0	3	Basic Data	237.27 GB	Basic Data
1	1	Basic Data	931.51 GB	Basic Data

Use when:

- You want partition table and layout details.
- Useful for troubleshooting disk issues.

4 Get-PSDrive

Get PowerShell drives in the system.

```
Get-PSDrive
```

Output (Sample)

Name	Used (GB)	Free (GB)	Provider	Root
C	154.92	82.45	FileSystem	C:\
D	605.38	326.12	FileSystem	D:\
E	341.95	123.80	FileSystem	E:\
F	7.70	7.20	FileSystem	F:\
Registry	–	–	Registry	HKLM:\
Env	–	–	Environment	Env:\

Use when:

- You want to see all drives (FileSystem, Registry, Env).
- Includes used and free space information.

5 Useful Examples

1. List all disks

```
Get-Disk | Format-Table
Number, FriendlyName, Size
```

2. List volumes with free space

```
Get-Volume | Select DriveLetter,
FileSystem, Label, SizeRemaining
```

3. Get partitions for Disk 0

```
Get-Partition -DiskNumber 0 |
Format-Table PartitionNumber, Type, Size
```

4. Show drives with free space > 10GB

```
Get-PSDrive -PSProvider FileSystem |
Where-Object { $_.Free -gt 10GB } |
Select Name, Used, Free
```

Example Output (Get-Volume)

DriveLetter	FileSystem	Label	Size	SizeRemaining
C	NTFS	Windows	237.37 GB	82.45 GB
D	NTFS	Data	931.50 GB	326.12 GB

Example Output (Get-PSDrive -PSProvider FileSystem)

Name	Used (GB)	Free (GB)	Provider
C	154.92	82.45	FileSystem
D	605.38	326.12	FileSystem

Disk Usage (Graphical View)

6 Combine Commands

Get disk and volume together

```
Get-Disk | Get-Partition | Get-Volume |
Select DiskNumber, DriveLetter, FileSystem, Size
```

Get only NTFS volumes

```
Get-Volume | Where-Object FileSystem -eq 'NTFS'
```

Remote disk information

```
Get-Disk -CimSession Server01
```

Quick Reference Table

Command	Purpose
Get-Disk	Get physical disk details
Get-Volume	Get logical volume details
Get-Partition	Get partition table details
Get-PSDrive	Get PowerShell drives
Get-Help	Get help for commands

Note

- Some commands need administrator privileges.
- Be careful while modifying disks or partitions.
- Always backup important data before making changes.
- Use on test systems before applying in production.

Disk Info Workflow

Summary

These PowerShell commands help you get complete disk, volume, partition and drive information quickly and easily. Perfect for system admins, developers and IT professionals.

Get-Disk • Get-Volume • Get-Partition • Get-PSDrive | PowerShell Makes Storage Management Easy! | Monitor. Manage. Optimize.

NETWORK INFORMATION

Manage and Get Network Details Easily in PowerShell!

Network Connectivity	IP Address IPv4 / IPv6	Adapter Network Device	Gateway Default Route	DNS Name Server	Firewall Status	Internet Connection	Monitor Traffic

Why Network Information?

- ✔ Check IP address and network settings
- ✔ Troubleshoot connectivity issues
- ✔ Monitor adapters and their status
- ✔ Find gateways, DNS and routes
- ✔ Useful for system admin & IT support

Syntax

```
Get-NetIPaddress
Get-NetAdapter
Get-NetIPConfiguration
Get-NetRoute
```

Key Points

- Get-NetIPaddress shows all IP addresses.
- Get-NetAdapter shows network adapters.
- Get-NetIPConfiguration shows IP configuration.
- Get-NetRoute shows routing table.
- Helps in network troubleshooting & monitoring.

Common Parameters

Parameter	Description
-InterfaceAlias	Filter by interface alias (name)
-AddressFamily	IPv4 or IPv6
-Detailed	Show detailed information
-IncludeAllCompartments	Include all routing compartments
-PolicyStore	View persistent or active store

Pro Tip

- Use -Detailed for complete information.
- Check adapter status before troubleshooting.
- Verify IP, Gateway, DNS for connectivity issues.
- Use Get-NetRoute to check default route.

1 Get-NetIPAddress

Get IP addresses of all network interfaces.

```
Get-NetIPaddress
```

Output (Sample)

InterfaceAlias	IPAddress	AddressFamily	PrefixLength
Ethernet	192.168.1.10	IPv4	24
Wi-Fi	192.168.0.25	IPv4	24
Loopback Pseudo-Interface 1	127.0.0.1	IPv4	8
Ethernet	fe80::1a2b:3c4d:5e6f:7890	IPv6	64
...	...		

Use when:

- You want to see IP addresses on all interfaces.
- You want to check IPv4 and IPv6 addresses.
- Troubleshoot IP configuration issues.

2 Get-NetAdapter

Get information about network adapters.

```
Get-NetAdapter
```

Output (Sample)

Name	InterfaceDescription	Status	MacAddress	LinkSpeed
Ethernet	Realtek PCIe GbE Family...	Up	00-1A-2B-3C-4D-5E	1 Gbps
Wi-Fi	Intel(R) Wi-Fi 6 AX201...	Up	00-1A-2B-3C-4D-5F	866 Mbps
Bluetooth Network Connection	Bluetooth Device...	Up	00-1A-2B-3C-4D-60	3 Mbps
vEthernet (Default Switch)	Hyper-V Virtual Ethernet...	Up	00-1A-2B-3C-4D-61	1 Gbps
...	...			

Use when:

- You want to see network adapters and status.
- You want to check adapter speed and MAC address.
- Troubleshoot adapter connectivity problems.

3 Get-NetIPConfiguration

Get IP configuration for all network interfaces.

```
Get-NetIPConfiguration
```

Output (Sample)

InterfaceAlias	IPv4Address	IPv4DefaultGateway	DNSServer
Ethernet	192.168.1.10	192.168.1.1	8.8.8.8 8.8.4.4
Wi-Fi	192.168.0.25	192.168.0.1	1.1.1.1 1.0.0.1
Loopback Pseudo-Interface 1	127.0.0.1	(none)	(none)
...	...		

Use when:

- You want to see IP, Gateway and DNS together.
- Check complete IP configuration of interfaces.
- Verify network settings at one place.

4 Get-NetRoute

Get routing table information.

```
Get-NetRoute
```

Output (Sample)

DestinationPrefix	NextHop	InterfaceAlias	RouteMetric
0.0.0.0/0	192.168.1.1	Ethernet	25
192.168.1.0/24	On-link	Ethernet	281
192.168.0.0/24	On-link	Wi-Fi	281
::/0	fe80::1%12	Ethernet	25
...	...		

Use when:

- You want to see routing table and gateways.
- Troubleshoot network and internet issues.
- Check default route and specific routes.

5 More Useful Commands & Examples

Get-NetAdapter | Where-Object Status -eq 'Up'

Show only active (Up) adapters.

```
Get-NetAdapter | Where-Object Status -eq 'Up'
```

Output (Sample)

Name	Status	LinkSpeed
Ethernet	Up	1 Gbps
Wi-Fi	Up	866 Mbps

Get-NetIPAddress -AddressFamily IPv4

Show only IPv4 addresses.

```
Get-NetIPAddress -AddressFamily IPv4
```

Output (Sample)

InterfaceAlias	IPAddress	PrefixLength
Ethernet	192.168.1.10	24
Wi-Fi	192.168.0.25	24

Get-NetIPConfiguration | Format-List

Detailed IP configuration.

```
Get-NetIPConfiguration | Format-List
```

Output (Sample)

```
InterfaceAlias      : Ethernet
IPv4Address         : 192.168.1.10
IPv4DefaultGateway  : 192.168.1.1
DNSServer           : {8.8.8.8, 8.8.4.4}
NetBIOS             : Enabled
...
```

Get-NetRoute -DestinationPrefix '0.0.0.0/0'

Show default gateway route.

```
Get-NetRoute -DestinationPrefix '0.0.0.0/0'
```

Output (Sample)

DestinationPrefix	NextHop	InterfaceAlias
0.0.0.0/0	192.168.1.1	Ethernet

Test-NetConnection 8.8.8.8

Test connectivity to 8.8.8.8.

```
Test-NetConnection 8.8.8.8
```

Output (Sample)

```
ComputerName      : 8.8.8.8
RemoteAddress     : 8.8.8.8
PingSucceeded     : True
TcpTestSucceeded  : True
```

Quick Reference Table

Command	Purpose
Get-NetIPaddress	Get IP addresses
Get-NetAdapter	Get network adapters
Get-NetIPConfiguration	Get IP configuration
Get-NetRoute	Get routing table
Test-NetConnection	Test connectivity
Get-DnsClientServerAddress	Get DNS servers

DNS Commands

- Get-DnsClientServerAddress — Show DNS servers
- Get-DnsClientCache — Show DNS cache
- Clear-DnsClientCache — Clear DNS cache

Note

- Some commands require Administrator privileges.
- IP changes may require network refresh.
- Use Test-NetConnection for deeper analysis.

Network Troubleshooting Workflow

Check Adapters Get-NetAdapter → Check IP Address Get-NetIPAddress → Check Configuration Get-NetIPConfiguration → Check Routes Get-NetRoute → Test Connectivity Test-NetConnection → Monitor & Fix Resolve Issues

Summary

These PowerShell commands help you get complete network information quickly and easily. Perfect for system admins, developers and IT professionals.

Check. Monitor. Troubleshoot. PowerShell Makes Networking Easy! 🚀 | Get-NetIPAddress • Get-NetAdapter • Get-NetIPConfiguration • Get-NetRoute | Stay Connected. Stay Ahead!

NETWORK TESTING

Test Network Connectivity, Ports, DNS & More Easily in PowerShell!

Ping ICMP Test	Port Test TCP/UDP	DNS Resolution	Connectivity Check	Network Status	Troubleshoot Issues	Analyze Results	Info Details

Why Network Testing?

- ✓ Check if a host is reachable
- ✓ Test ports and services
- ✓ Verify DNS resolution
- ✓ Troubleshoot network problems
- ✓ Useful for admins, developers & IT support

Syntax

```
Test-Connection
Test-NetConnection
Resolve-DnsName
Ping
```

Key Points

- Test-Connection is like Ping (ICMP test).
- Test-NetConnection checks ports & connectivity.
- Resolve-DnsName resolves domain names.
- Useful for troubleshooting network issues.
- Supports IPv4 and IPv6.

Common Parameters

Parameter	Description
-ComputerName	Specify target host or IP
-Count	Number of test packets
-Port	Port number for connection test
-InformationLevel	Detail level (Quiet, Detailed)
-DnsOnly	Resolve DNS only
-WarningAction	Action on warnings

Pro Tip

- Use Test-Connection for quick reachability check.
- Use Test-NetConnection to test specific ports.
- Use Resolve-DnsName to check DNS records.
- Use -Quiet for clean script output.

Note

- Some tests may require administrator privileges.
- Firewalls may block ICMP or ports.
- Use IPv4/IPv6 as required.
- Always verify connectivity from the target network.

1 Test-Connection (Ping)

Test network connectivity using ICMP (Ping).

```
Test-Connection google.com -Count 4
```

Output (Sample)

ComputerName	IP	ReplyTime (ms)	Status
google.com	142.250.192.14	21	Success
google.com	142.250.192.14	22	Success
google.com	142.250.192.14	20	Success
google.com	142.250.192.14	21	Success

Use when:

- You want to check if a host is reachable.
- You want to test basic network connectivity.

2 Test-NetConnection

Test TCP connection to a computer and port.

```
Test-NetConnection google.com -Port 80
```

Output (Sample)

```
ComputerName       : google.com
RemoteAddress      : 142.250.192.14
RemotePort         : 80
InterfaceAlias     : Wi-Fi
SourceAddress      : 192.168.1.25
TcpTestSucceeded   : True
```

Use when:

- You want to test if a port is open.
- You want detailed connection information.

3 Resolve-DnsName

Resolve DNS name to IP address or other records.

```
Resolve-DnsName google.com
```

Output (Sample)

Name	Type	TTL	Data
google.com	A	299	142.250.192.14
google.com	A	299	142.250.192.46
google.com	AAAA	299	2404:6800:4009:80d::200e
google.com	AAAA	299	2404:6800:4009:80d::200e

Use when:

- You want to resolve domain names.
- You want to check DNS records.

4 Ping (Alias for Test-Connection)

Ping is alias of Test-Connection.

```
Ping 8.8.8.8 -n 4
```

Output (Sample)

Address	Reply	TTL	Status
8.8.8.8	Reply from 8.8.8.8: bytes=32 time=18ms TTL=116		Success
8.8.8.8	Reply from 8.8.8.8: bytes=32 time=17ms TTL=116		Success
8.8.8.8	Reply from 8.8.8.8: bytes=32 time=18ms TTL=116		Success
8.8.8.8	Reply from 8.8.8.8: bytes=32 time=19ms TTL=116		Success

Use when:

- You prefer traditional Ping command.
- You want quick ICMP connectivity check.

5 Common Examples

Ping Google (4 times)

```
Test-Connection
google.com -Count 4
```

Result: Reachable ✓

Test Port 443 (HTTPS)

```
Test-NetConnection
google.com -Port 443
```

Result: TcpTestSucceeded : ✓

DNS Lookup

```
Resolve-DnsName
microsoft.com
```

Result: Returns IP addresses ✓

Test Port 22 (SSH)

```
Test-NetConnection
192.168.1.10 -Port 22
```

Result: TcpTestSucceeded : : False ✗

Ping with IPv6

```
Test-Connection
ipv6.google.com -Count 4
```

Result: Reachable (IPv6) ✓

6 More Useful Tests

Task	Command
Check Multiple Ports	Test-NetConnection google.com -Port 80,443,8080
Detailed Information	Test-NetConnection google.com -Port 443 -InformationLevel Detailed
Quiet Mode (Script)	Test-NetConnection google.com -Port 443 -InformationLevel Quiet
Check UDP Port	Test-NetConnection 8.8.8.8 -Port 53 -Udp
DNS Only Lookup	Resolve-DnsName google.com -DnsOnly

Quick Reference Table

Command	Purpose
Test-Connection	Test ICMP connectivity (Ping)
Test-NetConnection	Test TCP/UDP port connectivity
Resolve-DnsName	Resolve DNS names & records
Ping	Alias of Test-Connection
Get-NetIPConfiguration	Get IP configuration details
Get-NetRoute	View routing table

Sample Output: Test-NetConnection (Detailed)

```
ComputerName            : google.com
RemoteAddress           : 142.250.192.14
RemotePort              : 443
InterfaceAlias          : Wi-Fi
SourceAddress           : 192.168.1.25
TcpTestSucceeded        : True
PingSucceeded           : True
PingReplyDetails (RTT)  : 18 ms
```

Sample Output: Resolve-DnsName (Records)

Name	Type	TTL	Data
microsoft.com	A	300	13.107.246.45
microsoft.com	AAAA	300	2603:1026:20::45
microsoft.com	MX	3600	microsoft-com.mail.protection.outlook.com
microsoft.com	NS	172800	ns1-204.azure-dns.com
microsoft.com	NS	172800	ns2-204.azure-dns.net

...

DNS Record Types Quick View

- A : IPv4 Address
- AAAA : IPv6 Address
- MX : Mail Exchange
- NS : Name Server
- CNAME : Canonical Name
- TXT : Text Record

Common Ports

Port	Purpose
22	SSH
53	DNS (UDP/TCP)
80	HTTP
443	HTTPS
3389	RDP

Summary

These PowerShell commands help you test network connectivity, ports, DNS and troubleshoot network issues quickly and easily. Essential for IT professionals.

Network Testing Workflow

Choose Command (Ping / Port / DNS) → Specify Target (Host / IP / Domain) → → Check Results → Troubleshoot (If Failed) → Done! (Connected)

Test. Verify. Connect. | Test-Connection • Test-NetConnection • Resolve-DnsName • Ping | PowerShell Makes Network Testing Easy!

FIREWALL

Manage Firewall Rules Easily in PowerShell! 🔥

Create, View, Edit & Remove Firewall Rules with Simple Commands.

Firewall Protection | Rules Management | Allow Traffic | Block Traffic | Inbound Rules | Outbound Rules | Security Control | Monitor Status | Info Details

Why Use PowerShell Firewall Cmdlets?

- ✔ Manage Windows Firewall easily
- ✔ Create custom rules quickly
- ✔ Allow or block applications & ports
- ✔ Automate firewall configuration
- ✔ Useful for admins, developers & IT pros

Syntax

```
Get-NetFirewallRule
New-NetFirewallRule
Set-NetFirewallRule
Remove-NetFirewallRule
```

Key Points

- Get-NetFirewallRule shows existing rules.
- New-NetFirewallRule creates a new rule.
- Set-NetFirewallRule modifies an existing rule.
- Remove-NetFirewallRule deletes a rule.
- Rules can Allow or Block traffic.

Common Parameters

Parameter	Description
-DisplayName	Friendly name of the rule
-Direction	Inbound or Outbound
-Action	Allow or Block
-Protocol	TCP, UDP, ICMP, Any
-LocalPort	Local port number (e.g. 80)
-Program	Path to application
-RemoteAddress	Remote IP or range
-Enabled	True (Enable) or False (Disable)

Pro Tip

- Use -DisplayName to easily identify rules.
- Always specify -Direction (Inbound/Outbound).
- Use -Protocol and -LocalPort for port rules.
- Test rules after creating or modifying.

1 Get-NetFirewallRule

Get existing firewall rules.

```
Get-NetFirewallRule
```

Output (Sample)

DisplayName	Direction	Action	Enabled	Profile
Core Networking - DNS (UDP-In)	Inbound	Allow	True	Domain
File and Printer Sharing (SMB-In)	Inbound	Allow	True	Domain
Remote Desktop (TCP-In)	Inbound	Allow	False	Domain
Block Telnet	Inbound	Block	True	Any
...				

Use when:

- You want to list all firewall rules.
- You want to find a specific rule.
- You want to check rule status.

2 New-NetFirewallRule

Create a new firewall rule.

```
New-NetFirewallRule -DisplayName "Allow HTTP" -Direction Inbound `
-Action Allow -Protocol TCP -LocalPort 80
```

Output (Sample)

```
Name        : {a1b2c3d4-1111-2222-3333-444455556666}
DisplayName : Allow HTTP
Direction   : Inbound
Action      : Allow
Protocol    : TCP
LocalPort   : 80
Enabled     : True
Profile     : Any
...
```

Use when:

- You want to allow/block an application or port.
- You want to create custom security rules.
- You want to open a port for inbound/outbound traffic.

3 Set-NetFirewallRule

Modify an existing firewall rule.

```
Set-NetFirewallRule -DisplayName "Allow HTTP" -Enabled False
Set-NetFirewallRule -DisplayName "Allow HTTP" -Action Block
```

Output (Sample)

```
Name        : {a1b2c3d4-1111-2222-3333-444455556666}
DisplayName : Allow HTTP
Direction   : Inbound
Action      : Block
Protocol    : TCP
LocalPort   : 80
Enabled     : False
Profile     : Any
...
```

Use when:

- You want to enable or disable a rule.
- You want to change action (Allow/Block).
- You want to update rule properties.

4 Remove-NetFirewallRule

Remove (delete) a firewall rule.

```
Remove-NetFirewallRule -DisplayName "Allow HTTP"
Remove-NetFirewallRule -Name {a1b2c3d4-1111-2222-3333-4444}
```

Output (Sample)

```
Name        : {a1b2c3d4-1111-2222-3333-444455556666}
DisplayName : Allow HTTP
Removed     : True
...
```

Use when:

- You want to delete a rule no longer needed.
- You want to clean up old or unused rules.
- You want to remove rules created for testing.

5 Practical Examples

Allow HTTP (Port 80)

```
New-NetFirewallRule -DisplayName "Allow HTTP" `
-Direction Inbound -Action Allow -Protocol TCP -LocalPort 80
```

Block Telnet (Inbound)

```
New-NetFirewallRule -DisplayName "Block Telnet" `
-Direction Inbound -Action Block -Protocol TCP -LocalPort 23
```

Allow Outbound DNS

```
New-NetFirewallRule -DisplayName "Allow DNS Out" `
-Direction Outbound -Action Allow -Protocol UDP -RemotePort 53
```

Allow Application

```
New-NetFirewallRule -DisplayName "Allow Notepad" `
-Direction Inbound -Action Allow -Program "C:\Windows\System32\notepad.exe"
```

Disable Rule

```
Set-NetFirewallRule -DisplayName "Allow HTTP" -Enabled False
```

Remove Rule

```
Remove-NetFirewallRule -DisplayName "Allow HTTP"
```

6 Quick Reference Table

Command	Purpose
Get-NetFirewallRule	Get existing firewall rules
New-NetFirewallRule	Create a new firewall rule
Set-NetFirewallRule	Modify an existing rule
Remove-NetFirewallRule	Remove (delete) a rule

Common Actions

Allow Traffic	Permit network traffic
Block Traffic	Deny network traffic
Enable Rule	Turn rule on
Disable Rule	Turn rule off
Remove Rule	Delete rule permanently

Common Protocols

TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol
Any	All protocols

Note

- Run PowerShell as Administrator.
- Wrong rules can block network access.
- Backup existing rules before major changes.
- Use Remove-NetFirewallRule carefully.

Firewall Rule Workflow

1. Get Rules (Get-NetFirewallRule) → 2. Create Rule (New-NetFirewallRule) → 3. Modify Rule (Set-NetFirewallRule) → 4. Test Rule (Verify Traffic) → 5. Remove Rule (Remove-NetFirewallRule)

Summary

PowerShell Firewall cmdlets make it easy to manage Windows Firewall rules. Create, modify, allow or block traffic with just a few simple commands. Automate and secure your network!

Manage. Secure. Protect. | Get-NetFirewallRule • New-NetFirewallRule • Set-NetFirewallRule • Remove-NetFirewallRule | PowerShell Makes Firewall Management Easy! 🔥

SCHEDULED TASKS

Manage, Create & Automate Tasks Easily in PowerShell!

Automate Jobs • Save Time • Increase Productivity

Task Management | Create Task | Start Task | Disable Task | Enable Task | Schedule Triggers | Actions & Settings | Automation & Monitoring | Info Details

Why Use Scheduled Tasks?

- ✔ Automate repetitive jobs & scripts
- ✔ Run tasks at specific time or interval
- ✔ Run scripts even when you are not logged in
- ✔ Manage tasks locally or on remote systems
- ✔ Perfect for system maintenance & automation

Syntax

```
Get-ScheduledTask
New-ScheduledTask
Start-ScheduledTask
Disable-ScheduledTask
Enable-ScheduledTask
```

Key Points

- Get-ScheduledTask lists all scheduled tasks.
- New-ScheduledTask creates a new scheduled task.
- Start-ScheduledTask runs a task immediately.
- Disable-ScheduledTask disables a task.
- Enable-ScheduledTask enables a disabled task.

Common Parameters

Parameter	Description
-TaskName	Name of the task
-TaskPath	Path of the task (e.g. \\MyTasks\)
-Action	Action to perform (Execute a program)
-Trigger	When the task should run
-User	User account to run the task
-RunLevel	Privilege level (Highest / Limited)
-Force	Overwrite if task already exists

Pro Tip

- Use -TaskPath to organize tasks in folders.
- Use -RunLevel Highest for admin tasks.
- Test your task after creating.
- Always check task history for issues.

Note

- Some tasks need administrator privileges.
- Incorrect scripts may affect system performance.
- Check task history if task not running.
- Use Try-Catch in scripts for better reliability.

1 Get-ScheduledTask

Get all scheduled tasks.

```
Get-ScheduledTask
```

Output (Sample)

TaskName	TaskPath	State	Actions
DailyBackup	\\Backup	Ready	1
CleanTemp	\\Maintenance	Running	1
MyScriptTask	\\Scripts	Ready	1
UpdateCheck	\\System	Disabled	1
...			

Use when:

- You want to list all tasks.
- You want to check task status.

2 New-ScheduledTask

Create a new scheduled task.

```
New-ScheduledTask -Action $action
-Trigger $trigger -TaskName "MyTask"
```

Output (Sample)

```
TaskName   :  MyTask
TaskPath   :  \\MyTasks\
State      :  Ready
Actions    :  1
Triggers   :  1
Author     :  DESKTOP-7Q9K2V1\User
Date       :  25-05-2024 10:30:00
...
```

Use when:

- You want to create a new scheduled task.
- You want to automate a script or program.

3 Start-ScheduledTask

Start a scheduled task.

```
Start-ScheduledTask -TaskName "MyTask"
```

Output (Sample)

```
TaskName    :  MyTask
State       :  Running
Result      :  The operation completed
               successfully.
Start Time  :  25-05-2024 10:30:15
```

Use when:

- You want to run a task immediately.
- You want to test a task.

4 Disable-ScheduledTask

Disable a scheduled task.

```
Disable-ScheduledTask -TaskName "MyTask"
```

Output (Sample)

```
TaskName  :  MyTask
State     :  Disabled
Result    :  The operation completed successfully.
```

Use when:

- You want to stop a task from running.
- You want to disable a task temporarily.

5 Enable-ScheduledTask

Enable a disabled scheduled task.

```
Enable-ScheduledTask -TaskName "MyTask"
```

Output (Sample)

```
TaskName  :  MyTask
State     :  Ready
Result    :  The operation completed successfully.
```

Use when:

- You want to enable a disabled task.
- You want the task to run again.

Examples

List All Tasks

```
Get-ScheduledTask
```

Get all scheduled tasks

Create Daily Task (Run Notepad at 9 AM)

```
$action = New-ScheduledTaskAction -Execute "notepad.exe"
$trigger = New-ScheduledTaskTrigger -Daily -At 9am
New-ScheduledTask -TaskName "DailyNotepad" -Action $action `
-Trigger $trigger -Description "Open Notepad daily at 9 AM"
```

Start Task

```
Start-ScheduledTask -TaskName "DailyNotepad"
```

Run the task now

Disable Task

```
Disable-ScheduledTask -TaskName "DailyNotepad"
```

Disable the task

Enable Task

```
Enable-ScheduledTask -TaskName "DailyNotepad"
```

Enable the task

Remove Task

```
Unregister-ScheduledTask -TaskName "DailyNotepad" -Confirm:$false
```

Remove a task permanently

Triggers (Common)

Trigger Type	Description
-Once	Run once at a specific time
-Daily	Run daily
-Weekly	Run weekly on specific days
-AtStartup	Run at system startup
-AtLogOn	Run when user logs on
-OnIdle	Run when system is idle

Actions (Common)

Action Type	Description
-Execute	Run a program or script
-Argument	Pass arguments to program
-WorkingDirectory	Set working directory
-ErrorAction	Action if error occurs

Quick Reference Table

Command	Purpose
Get-ScheduledTask	List all tasks
New-ScheduledTask	Create a new task
Start-ScheduledTask	Start a task now
Disable-ScheduledTask	Disable a task
Enable-ScheduledTask	Enable a task
Unregister-ScheduledTask	Remove a task

Scheduled Task Workflow

1. Plan Task: Decide what & when to run →
2. Create Task: Use New-ScheduledTask →
3. Schedule: Set Trigger (time/condition) →
4. Start & Test: Run task to verify →
5. Manage Task: Enable / Disable / Modify →
6. Monitor: Check status & history

Task History & Logs

Check task history using:

```
Get-WinEvent -LogName Microsoft-Windows-TaskScheduler/Operational
```

Helps you troubleshoot and verify task execution.

Summary

PowerShell Scheduled Task cmdlets help you automate and manage tasks easily. Save time, improve productivity and keep your system running smoothly.

Automate Today. Save Time Tomorrow. | Get-ScheduledTask • New-ScheduledTask • Start-ScheduledTask • Disable-ScheduledTask • Enable-ScheduledTask | PowerShell Makes Automation Easy!

REGISTRY in PowerShell

Manage Windows Registry Keys, Values & Configuration Easily!

- Registry Database
- Key (Folder)
- Value (Data)
- String REG_SZ
- DWORD REG_DWORD
- QWORD REG_QWORD
- Multi-String REG_MULTI_SZ
- Binary REG_BINARY
- Configuration Settings
- Security Permissions

Why Use Registry Cmdlets?

- ✔ Read and modify registry settings
- ✔ Automate system configuration
- ✔ Manage application settings
- ✔ Backup and restore registry data
- ✔ Perfect for admins and IT pros

Syntax

```
Get-ItemProperty
Set-ItemProperty
New-ItemProperty
Remove-ItemProperty
```

Key Points

- Registry stores system and application settings.
- Keys are like folders, values store data.
- Always backup before making changes.
- Use correct data type for values.
- Requires Administrator privileges for most changes.

Common Parameters

Parameter	Description
-Path	Registry key path
-Name	Name of the value
-Value	Value data
-PropertyType	Type of the value
-Force	Force creation/removal
-ErrorAction	Define error handling

Pro Tip

- Use -LiteralPath if key path contains special chars.
- Export registry key before making changes.
- Use correct data type for best results.
- Test in a safe environment first.

1 Get-ItemProperty

Get properties (values) of a registry key.

```
Get-ItemProperty -Path "HKLM:\\Software\\Microsoft\
\Windows\\CurrentVersion\\Run"
```

Output (Sample)

PSPath	Property	Type	Data
HKLM:\...Run	(Default)	String	(value not set)
HKLM:\...Run	OneDrive	String	"C:\Program Files..."
HKLM:\...Run	SecurityHealth	String	%windir%\system32...

Use when:

- You want to read registry key values.
- You want to list all values under a key.

2 Set-ItemProperty

Set or update the value of a registry key.

```
Set-ItemProperty -Path "HKLM:\\Software\\MyApp"
-Name "InstallPath" -Value "C:\MyApp" -Type String
```

Output (Sample)

PSPath	Name	Value	Type
HKLM:\...MyApp	InstallPath	C:\MyApp	String

Use when:

- You want to update an existing value.
- You want to change a registry setting.

3 New-ItemProperty

Create a new value in a registry key.

```
New-ItemProperty -Path "HKCU:\\Software\\MyApp"
-Name "Theme" -Value "Dark" -PropertyType String
```

Output (Sample)

PSPath	Name	Type	Data
HKCU:\.....MyApp	Theme	String	Dark

Use when:

- You want to create a new registry value.
- You want to add new configuration data.

4 Remove-ItemProperty

Remove a value from a registry key.

```
Remove-ItemProperty -Path "HKCU:\\Software\\MyApp"
-Name "Theme"
```

Output (Sample)

PSPath	Name	Result
HKCU:\....MyApp	Theme	Deleted

Use when:

- You want to delete a registry value.
- You want to clean up old or unused settings.

Common Registry Locations

Hive	Description
HKLM:\	HKEY_LOCAL_MACHINE (All Users)
HKCU:\	HKEY_CURRENT_USER (Current User)
HKCR:\	HKEY_CLASSES_ROOT (File Associations)
HKU:\	HKEY_USERS (All User Profiles)
HKCC:\	HKEY_CURRENT_CONFIG (Current Hardware)

Value Types

Type	Description	Example
String (REG_SZ)	Text data (default)	"C:\Path\File.exe"
ExpandString (REG_DWMD_SZ)	String with environment variables	%SystemRoot%\system32
DWORD (REG_DWORD)	32-bit number	1, 0, 123456
QWORD (REG_QWORD)	64-bit number	1234567890123
MultiString (REG_MULTI_SZ)	Multiple strings	"one\0two\0three\0\0"
Binary (REG_BINARY)	Binary data	01 02 FF 1A ...

Practical Examples

`Get-ItemProperty -Path "HKLM:\\Software\\MyApp"`	Get all values of a key
`Set-ItemProperty -Path "HKCU:\\Control Panel\\Desktop" -Name "Wallpaper" -Value "C:\Wall.jpg"`	Set wallpaper path
`New-ItemProperty -Path "HKCU:\\Software\\MyApp" -Name "Version" -Value "1.0" -PropertyType String`	Create new value
`Remove-ItemProperty -Path "HKCU:\\Software\\MyApp" -Name "OldValue"`	Remove a value
`Get-ChildItem -Path "HKLM:\\Software"`	List subkeys
`New-Item -Path "HKCU:\\Software\\MyApp" -Force`	Create new key

Quick Reference Table

Command	Purpose
Get-ItemProperty	Read registry values
Set-ItemProperty	Update registry values
New-ItemProperty	Create new registry value
Remove-ItemProperty	Delete registry value
New-Item	Create new registry key
Remove-Item	Delete registry key
Get-ChildItem	List subkeys
Export-Registry	Export registry key
Import-Registry	Import registry file

Note

- Incorrect changes can affect system stability.
- Always run PowerShell as Administrator.
- Backup registry before making major changes.
- Be careful with system-critical keys.

Backup & Restore

Export Registry Key

```
Export-Registry -Path "HKCU:\\Software\\MyApp"
-FilePath "C:\Backup\\MyApp.reg"
```

Export a key to .reg file

Import Registry Key

```
Import-Registry -FilePath
"C:\Backup\\MyApp.reg"
```

Import a .reg file to restore

Registry Workflow

1. Identify Key → Find the registry key
2. Read/Check → Get current values
3. Modify/Create → Set or create values
4. Verify → Verify changes
5. Backup → Export key for backup

Summary

PowerShell registry cmdlets give you full control over Windows Registry. Read, create, update, and delete keys and values easily. Automate configuration and manage system settings like a pro!

Read. Modify. Automate. | Get-ItemProperty • Set-ItemProperty • New-ItemProperty • Remove-ItemProperty | PowerShell Makes Configuration Easy!

WINDOWS EVENT LOGS

View, Filter & Analyze Windows Event Logs Easily in PowerShell!

Monitor • Troubleshoot • Audit • Secure • Diagnose

Event Log Viewer | System Logs | Application Logs | Security Logs | Event ID Filter | Logs Search | Diagnostics & Analysis | Monitoring & Alerts

Why Use Get-WinEvent?

- ✓ Access all Windows event logs
- ✓ Filter by Log, Level, ID, Time & more
- ✓ Powerful querying and filtering
- ✓ Export logs for reporting & analysis
- ✓ Perfect for troubleshooting & auditing

Syntax

```
Get-WinEvent
Get-WinEvent -LogName <LogName>
Get-WinEvent -FilterHashtable @{...}
Get-WinEvent -FilterXPath "<Query>"
```

Key Points

- Get-WinEvent retrieves events from Windows logs.
- Use -LogName to specify a log.
- Use -FilterHashtable for fast filtering.
- Use -MaxEvents to limit results.
- Use -Oldest to read oldest events first.
- Events have Id, Level, Time, Provider & Message.

Common Parameters

Parameter	Description
-LogName	Name of the event log
-FilterHashtable	Filter events using hashtable
-FilterXPath	Filter events using XPath query
-MaxEvents	Maximum number of events
-Oldest	Read events from oldest first
-StartTime	Start time of events
-EndTime	End time of events
-ProviderName	Filter by provider name
-Id	Filter by Event ID
-Level	Filter by level (1-5)

Pro Tip

- Use -FilterHashtable for best performance.
- Filter by Event ID to find specific issues.
- Export logs to .evtx or .csv for reports.
- Combine filters for more accurate results.

Note

- Some logs need Administrator privileges.
- Large queries may take time.
- Clear old logs to save disk space.
- Event logs are in .evtx format.

1 Get-WinEvent (All Events)

Get recent events from all logs.

```
Get-WinEvent -MaxEvents 10
```

Output (Sample)

TimeCreated	Id	LevelDisplayName	LogName	Message
25-05-2024 10:30:15	7036	Information	System	The Windows...
25-05-2024 10:30:12	10016	Warning	Application	Application...
25-05-2024 10:30:10	4624	Information	Security	An account...
...				

Use when:

- You want to see recent events.
- Quickly check what's happening in the system.

2 Get from Specific Log

Get events from a specific log.

```
Get-WinEvent -LogName System -MaxEvents 10
```

Output (Sample)

TimeCreated	Id	LevelDisplayName	LogName	Message
25-05-2024 10:30:15	7036	Information	System	The Windows...
25-05-2024 10:29:58	7040	Information	System	The start type...
25-05-2024 10:29:50	6005	Information	System	The Event Log...
...				

Use when:

- You want events from System, Application or Security.
- Helps in focused troubleshooting.

3 Filter by Level

Get events with specific level.

```
Get-WinEvent -LogName System -FilterHashtable @{Level=2}
```

Output (Sample)

TimeCreated	Id	LevelDisplayName	LogName	Message
25-05-2024 10:29:58	7040	Warning	System	The start type...
25-05-2024 10:28:40	10016	Warning	System	The Microsoft...
25-05-2024 10:27:22	6008	Error	System	The previous...
...				

Use when:

- You want only Error, Warning or Critical events.
- Level: 1=Critical, 2=Error, 3=Warning, 4=Information, 5=Verbose.

4 Filter by Event ID

Get events with specific Event ID.

```
Get-WinEvent -LogName Security -FilterHashtable @{Id=4624}
```

Output (Sample)

TimeCreated	Id	LevelDisplayName	LogName	Message
25-05-2024 10:30:10	4624	Information	Security	An account was...
25-05-2024 10:29:45	4624	Information	Security	An account was...
25-05-2024 10:29:20	4624	Information	Security	An account was...
...				

Use when:

- You want events with a specific Event ID.
- Useful for login, failure, shutdown, etc.

Common Event Logs

Log Name	Description
System	System events and services
Application	Application errors and info
Security	Security auditing events
Setup	Windows setup events
Microsoft-Windows-PowerShell/Operational	PowerShell operational logs
Microsoft-Windows-TaskScheduler/Operational	Task Scheduler events
Microsoft-Windows-DNS Client Events/Operational	DNS client events

Common Event Levels

Level	Value	Description	Color
Critical	1	Critical error, system down	
Error	2	Error events	
Warning	3	Warning events	
Information	4	Informational events	
Verbose	5	Detailed informational events	

Practical Examples

Last 20 events from System log

```
Get-WinEvent -LogName System -MaxEvents 20 -Newest
```

Events from last 1 hour

```
Get-WinEvent -LogName System -FilterHashtable @{StartTime=(Get-Date).AddHours(-1)}
```

Filter by ID and Level

```
Get-WinEvent -LogName System -FilterHashtable @{Id=6008;Level=2}
```

Export events to CSV

```
Get-WinEvent -LogName System -MaxEvents 100 | Export-Csv C:\Logs\system.csv -NoTypeInformation
```

Quick Reference Table

Command	Purpose
Get-WinEvent	Get recent events
-LogName	Specify log name
-FilterHashtable	Filter using hashtable
-FilterXPath	Filter using XPath
-MaxEvents	Limit number of events
-Oldest	Read oldest events first
-Id	Filter by Event ID
-Level	Filter by level
Export-Csv	Export to CSV file
Export-Clixml	Export to XML file

Event Log Analysis Workflow

1. Choose Log (System / Application / Security) → 2. Filter Events (Level / ID / Time) → 3. Analyze Results (Check Details & Message) → 4. Take Action (Troubleshoot / Fix) → 5. Export / Report (CSV / XML / Logs)

Summary

Get-WinEvent is a powerful cmdlet to read and analyze Windows event logs.
It helps in monitoring, troubleshooting, auditing and securing your system.
Use filters to find important events quickly and take action effectively!

Monitor. Diagnose. Secure. | Get-WinEvent • Event Logs • System • Application • Security • Diagnostics | PowerShell Makes Troubleshooting Easy!

USERS in PowerShell

Manage Local Users & Accounts Easily in PowerShell!

Create, Modify, Remove & Manage User Accounts with Simple Commands.

User Account | Create User | Manage Properties | Password Control | Administrator Rights | Group Membership | Account Status | Search Users | Info Details

Why Manage Users with PowerShell?

- ✔ Quickly manage local user accounts
- ✔ Automate user creation & configuration
- ✔ Set passwords, descriptions & properties
- ✔ Enable/disable accounts easily
- ✔ Perfect for system admins & IT pros

Syntax

```
Get-LocalUser
New-LocalUser
Set-LocalUser
Remove-LocalUser
```

Key Points

- Get-LocalUser lists all local users.
- New-LocalUser creates a new user account.
- Set-LocalUser modifies user properties.
- Remove-LocalUser deletes a user account.
- Use strong passwords & proper permissions.

Common Parameters

Parameter	Description
-Name	Specifies the user name
-Pasword	Sets the user password (SecureString)
-Description	Adds a description for the user
-FullName	Sets the full name
-AccountNeverExpires	Keeps account from expiring
-PasswordNeverExpires	Prevents password from expiring
-Disabled	Creates or sets account as disabled

Pro Tip

- Always use -Password (ConvertTo-SecureString).
- Add user to Administrators group if needed.
- Regularly review and manage user accounts.
- Use strong passwords for better security.

Note

- Some actions need Administrator privileges.
- Be careful while deleting user accounts.
- Disabled accounts can still exist on the system.
- Lost passwords require account reset.

1 Get-LocalUser

List all local user accounts.

```
Get-LocalUser
```

Output (Sample)

Name	Enabled	Description
Administrator	True	Built-in account for admin...
Guest	False	Built-in account for guest...
Lovejeet	True	Local user account
TechUser	True	Created for testing

Use when:

- You want to view all local users.
- You want to check user account status.

2 New-LocalUser

Create a new local user account.

```
$pwd = ConvertTo-SecureString "P@ssw0rd123" -AsPlainText -Force
New-LocalUser -Name "TechUser" -Password $pwd `
  -FullName "Tech User" -Description "Test account"
```

Output (Sample)

Name	Enabled	Description
TechUser	True	Test account

Use when:

- You want to create a new user.
- You want to set password & description.

3 Set-LocalUser

Modify properties of an existing user.

```
Set-LocalUser -Name "TechUser" -FullName "Tech User" `
  -Description "Updated user account" -PasswordNeverExpires $true
```

Output (Sample)

Name	FullName	PasswordNeverExpires	Description
TechUser	Tech User	True	Updated user account

Use when:

- You want to update user properties.
- You want to change password settings.

4 Remove-LocalUser

Remove (delete) a local user account.

```
Remove-LocalUser -Name "TechUser"
```

Output (Sample)

Name	Result
TechUser	Deleted

Use when:

- You want to delete a user account.
- You no longer need the user on system.

5 Disable/Enable User

Disable or enable a user account.

```
Disable-LocalUser -Name "TechUser"
Enable-LocalUser -Name "TechUser"
```

Output (Sample)

Name	Enabled
TechUser	False
TechUser	True

Use when:

- You want to temporarily disable a user.
- You want to enable a disabled account.

Common User Information

Property	Description
Name	User account name
Enabled	True if account is enabled
Description	Description of the user
FullName	Full name of the user
LastLogon	Last logon date & time
PasswordLastSet	Password last set date
PasswordExpires	Password expiration date

Change User Password

Set a new password for a user.

```
$pwd = ConvertTo-SecureString "N3w0@ss!" -AsPlainText -Force
Set-LocalUser -Name "Lovejeet" -Password $pwd
```

Use when:

- You want to change a user password.
- User forgot password (if you know the new one).

Add User to Administrators Group

Add a user to Administrators group.

```
Add-LocalGroupMember -Group "Administrators" -Member "Lovejeet"
```

Use when:

- You want to give admin rights to a user.
- User needs permission to install software.

Check User Membership

```
Get-LocalGroupMember -Group "Administrators"
```

Output (Sample)

Name	ObjectClass
Administrator	User
Lovejeet	User

Search User

Search user by name (like pattern).

```
Get-LocalUser -Name "*tech*"
```

Output (Sample)

Name	Enabled	Description
TechUser	True	Created for testing

Create User with Options (Example)

```
$pwd = ConvertTo-SecureString "P@ssw0rd123" -AsPlainText -Force
New-LocalUser -Name "DevUser" -Password $pwd `
  -FullName "Developer User" -Description "Developer Account" `
  -AccountNeverExpires $true -PasswordNeverExpires $true
```

Use when:

- You want to create a user with advanced options.
- You want account & password to never expire.

Disable User Account

```
Disable-LocalUser -Name "DevUser"
```

Output (Sample)

Name	Enabled
DevUser	False

Use when:

- You want to disable user without deleting.
- Useful for temporary suspension.

Delete User Account

```
Remove-LocalUser -Name "DevUser"
```

Output (Sample)

Name	Result
DevUser	Deleted

Use when:

- You want to permanently remove a user.
- User is no longer needed on the system.

Quick Reference Table

Command	Purpose
Get-LocalUser	List all local users
New-LocalUser	Create a new user
Set-LocalUser	Modify user properties
Remove-LocalUser	Delete a user
Disable-LocalUser	Disable a user
Enable-LocalUser	Enable a user
Add-LocalGroupMember	Add user to group
Get-LocalGroupMember	Show group members

Common User Accounts

Account	Description
Administrator	Built-in admin account
Guest	Built-in guest account
DefaultAccount	System managed account
WDAGUtilityAccount	Windows Defender account
YourCustomUser	User created by you

User Management Workflow

1. Create User — New-LocalUser → 2. Configure — Set-LocalUser → 3. Add to Group — Add-LocalGroupMember → 4. Manage Access — Enable/Disable User → 5. Review — Get-LocalUser → 6. Remove (if needed) — Remove-LocalUser

Summary

PowerShell makes user account management simple, fast & powerful.
Create, modify, disable, enable & remove users with just a few commands.
Automate tasks, save time & keep your system secure!

Manage Users. Secure System. Save Time. | Get-LocalUser • New-LocalUser • Set-LocalUser • Remove-LocalUser | PowerShell Gives You Full Control!

GROUPS in PowerShell

Manage Local Groups, Members & Permissions Easily!

Create, Manage & Control Groups with Simple PowerShell Commands.

Group | Create Group | Members | Add Member | Remove Member | Permissions | Users | Properties | Info

Why Manage Groups in PowerShell?

- ✔ Organize users with groups
- ✔ Simplify permission management
- ✔ Add or remove members easily
- ✔ Manage system & custom groups
- ✔ Perfect for admins & IT pros

Syntax

```
Get-LocalGroup
New-LocalGroup
Add-LocalGroupMember
Remove-LocalGroupMember
```

Key Points

- Get-LocalGroup lists all local groups.
- New-LocalGroup creates a new group.
- Add-LocalGroupMember adds users to a group.
- Remote-LocalGroupMember removes users.
- Groups help manage permissions easily.
- Built-in groups come by default in Windows.

Common Parameters

Parameter	Description
-Name	Specifies the group name
-Description	Sets the group description
-Member	Specifies member (user) name
-SID	Specifies security identifer
-Verbose	Shows detailed information
-ErrorAction	Defines error handling

Pro Tip

- Use groups to manage multiple users easily.
- Add users to groups, not directly to permissions.
- Check group membership before changes.
- Use built-in groups for common tasks.

Note

- Some groups need Administrator privileges.
- Built-in groups cannot be deleted.
- Removing the last admin can lock you out.
- Changes may require logoff/login.

1 Get-LocalGroup

List all local groups.

```
Get-LocalGroup
```

Output (Sample)

Name	Description
Administrators	Administrators have compl...
Users	Users are prevented from ...
Guests	Guests have the same acces...
Remote Desktop Users	Members in this group are...
...	

Use when:
- You want to view all local groups.
- You want to check group details.

2 New-LocalGroup

Create a new local group.

```
New-LocalGroup -Name "TechTeam"
-Description "IT Team Group"
```

Output (Sample)

Name	Description
TechTeam	IT Team Group

Use when:
- You want to create a new group.
- You want to organize users.

3 Add-LocalGroupMember

Add one or more users to a group.

```
Add-LocalGroupMember -Group "TechTeam" -Member "John"
```

Output (Sample)

Group	Member	Result
TechTeam	John	Success

Use when:
- You want to add users to a group.
- You want to give group permissions.

4 Remove-LocalGroupMember

Remove a user from a group.

```
Remove-LocalGroupMember -Group "TechTeam" -Member "John"
```

Output (Sample)

Group	Member	Result
TechTeam	John	Success

Use when:
- You want to remove users from a group.
- User no longer needs group access.

5 Get-LocalGroupMember

List members of a group.

```
Get-LocalGroupMember -Group "TechTeam"
```

Output (Sample)

ObjectClass	Name
User	John
User	Alex
User	Priya
...	

Use when:
- You want to see group members.
- Check who has access via group.

Common Built-in Groups

Group Name	Purpose
Administrators	Full control of the computer
Users	Standard users
Guests	Guest access to computer
Power Users	Advanced users (legacy)
Remote Desktop Users	Allow logon through RDP
Backup Operators	Backup and restore files
Event Log Readers	Read event logs from local machine

Group Properties

Property	Description
Name	Name of the group
Description	Description about the group
SID	Security Identifier
PrincipalSource	Where the group is stored
GroupCategory	Security / Distribution
Members	Members of the group

Check Group Existence (Example)

```
Get-LocalGroup -Name "TechTeam" -ErrorAction SilentlyContinue
```

Output (Sample)

Name	Description
TechTeam	IT Team Group

Use when:
- You want to check if a group exists.
- Useful in scripts & automation.

Rename Group (Example)

```
Rename-LocalGroup -Name "OldGroup" -NewName "NewGroup"
```

Output (Sample)

Old Name	New Name	Result
OldGroup	NewGroup	Success

Use when:
- You want to rename a group.
- Keep names meaningful.

Delete Group (Example)

```
Remove-LocalGroup -Name "OldGroup"
```

Output (Sample)

Name	Result
OldGroup	Deleted

Use when:
- You want to delete a group.
- Group is no longer needed.

List Members with Details

```
Get-LocalGroupMember -Group "TechTeam" | Select Name, ObjectClass, SID
```

Output (Sample)

Name	ObjectClass	SID
John	User	S-1-5-21-1001...1001
Alex	User	S-1-5-21-1001...1002
Priya	User	S-1-5-21-1001...1003

Add Multiple Members

```
Add-LocalGroupMember -Group "TechTeam" -Member "User1","User2","User3"
```

Output (Sample)

Group	Members	Result
TechTeam	User1, User2, User3	Success

Export Group Members to CSV

```
Get-LocalGroupMember -Group "TechTeam" | [illegible]
[illegible]TypeInformation
```

Output (Sample)

File saved: C:\GroupMembers.csv

Contains group members list.

Import Members from CSV (Example)

```
Import-Csv -Path "C:\Users.csv" | ForEach-Object {
Add-LocalGroupMember -Group "TechTeam" -Member $_.Name }
```

Output (Sample)

Import users from CSV and add to group.

Each row should have column: Name

Check Group Permissions (Example)

```
Get-Acl -Path "WinNT://./TechTeam" | Format-List
```

Output (Sample)

```
Path        : WinNT://!./TechTeam
Access      : Allow
Identity    : BUILTIN\Users
Rights      : ReadAndExecute
Inheritance : ContainerInherit
```

Group Management Workflow

1. Create Group — New-LocalGroup → 2. Add Members — Add-LocalGroupMember → 3. Verify Members — Get-LocalGroupMember → 4. Manage Permissions — Groups Control Access → 5. Remove Members — Remove-LocalGroupMember → 6. Delete Group (if needed) — Remove-LocalGroup

Summary

PowerShell makes group management simple & powerful. Create groups, add/remove members, and control permissions with just a few commands. Automate tasks, save time & keep your system secure!

Organize. Manage. Secure. | Get-LocalGroup • New-LocalGroup • Add-LocalGroupMember • Remove-LocalGroupMember | PowerShell Makes Group Management Easy!

PERMISSIONS in PowerShell

Manage File, Folder & Object Permissions Easily!

Secure • Control • Protect • Audit • Automate Permissions

Access Control | Permissions Management | ACL (Access Control List) | Owner Management | Security Settings | Audit Permissions | Files & Folders Protection | Info Details

Why Manage Permissions?

- ✔ Control who can access your data
- ✔ Protect sensitive files & folders
- ✔ Prevent unauthorized changes
- ✔ Comply with security policies
- ✔ Perfect for admins & IT pros

Syntax

```
Get-Acl -Path "<Path>"
Set-Acl -Path "<Path>" -AclObject <ACL>
icacls <Path> /grant <User>:<Rights>
icacls <Path> /deny <User>:<Rights>
```

Key Points

- Get-Acl gets the ACL (permissions) of an item.
- Set-Acl applies ACL to a file, folder or object.
- ACL (Access Control List) defines permissions.
- Permissions can be Allow or Deny.
- Owner can modify permissions and take ownership.
- Use icacls for advanced permission control.

Common Parameters

Parameter	Description
-Path	Path of the file/folder
-AclObject	ACL object to set
-Verbose	Show detailed output
-ErrorAction	Handle errors
-Recurse	Apply to subfolders (with icacls)
-Force	Force action if needed

Pro Tip

- Use Get-Acl to backup permissions before changes.
- Use icacls for bulk permission management.
- Always test permission changes carefully.
- Use -WhatIf (SupportsShouldProcess) for safe testing.

Note

- Changing permissions may block access.
- Take ownership before modifying restricted items.
- Some actions require Administrator privileges.
- Backup critical ACLs before applying changes.

1 Get-Acl (View Permissions)

Get the ACL (permissions) of a file or folder.

```
Get-Acl -Path "C:\Data\Report.txt"
```

Output (Sample)

```
Path    : Microsoft.PowerShell.Core\FileSystem::C:\Data\Report.txt
Owner   : DESKTOP-ABC\Admin
Group   : DESKTOP-ABC\Users
Access
FileSystemRights    AccessControlType    IdentityReference
FullControl         Allow                DESKTOP-ABC\Admin
Modify              Allow                DESKTOP-ABC\Users
Read & Execute      Allow                BUILTIN\Users
```

Use when:

- You want to view current permissions.
- You want to audit access on files/folders.

2 Set-Acl (Apply Permissions)

Set or modify ACL (permissions) on a file or folder.

```
$acl = Get-Acl -Path "C:\Data\Report.txt"
Set-Acl -Path "C:\Data\Report.txt" -AclObject $acl
```

Output (Sample)

```
Path      : C:\Data\Report.txt
AclObject : System.Security.AccessControl.FileSecurity
Status    : Success
```

Use when:

- You want to apply modified permissions.
- You want to restore saved ACL.

3 icacls (Grant Permissions)

Grant Allow permissions to a user or group.

```
icacls "C:\Data" /grant John:(OI)(CI)M
```

Output (Sample)

```
processed file: C:\Data
Successfully processed 1 files;
Failed processing 0 files
```

Use when:

- You want to give allow permissions.
- You want to grant access to a folder & subfolders.

4 icacls (Deny Permissions)

Deny permissions to a user or group.

```
icacls "C:\Data" /deny Guest:(OI)(CI)W
```

Output (Sample)

```
processed file: C:\Data
Successfully processed 1 files;
Failed processing 0 files
```

Use when:

- You want to deny access.
- You want to block write or modify access.

Common Access Rights

Right	Description
F	Full Control
M	Modify
RX	Read & Execute
R	Read
W	Write
D	Delete
WO	Write Owner
WDAC	Write DAC (Change Permissions)
S	Synchronize

AccessControlType

Type	Description
Allow	Permission is allowed
Deny	Permission is denied
Inherited	Permission inherited from parent
Audit	Permission for auditing

Example: Change Owner

```
$acl = Get-Acl -Path "C:\Data\Secret.txt"
$acl.SetOwner([System.Security.Principal.NTAccount]"John")
Set-Acl -Path "C:\Data\Secret.txt" -AclObject $acl
```

Use when:

- You want to take ownership of a file/folder.
- Current owner does not allow changes.

Example: Remove Permissions

```
icacls "C:\Data\Report.txt" /remove John
```

Output (Sample)

```
processed file: C:\Data\Report.txt
Successfully processed 1 files; Failed 0 files
```

Use when:

- You want to remove specific user permissions.
- You no longer want the user to have access.

Backup & Restore ACL

Backup ACL

```
Get-Acl -Path "C:\Data" | Out-File -FilePath "C:\Backup\acl.txt"
```

Restore ACL

```
Get-Content "C:\Backup\acl.txt" | Set-Acl -Path "C:\Data"
```

Use when:

- You want to backup permissions.
- You want to restore ACL after changes.

icacls Common Switches

Switch	Description
/grant	Grant permissions
/deny	Deny permissions
/remove	Remove permissions
/inheritance:r	Remove inherited permissions
/inheritance:e	Enable inheritance
/t	Apply to files in subdirectories
c	Continue on errors
q	Quiet mode

Example Scenarios

Scenario	Command
Give Modify to Users	icacls "C:\Data" /grant Users:(OI)(CI)M
Read access to Everyone	icacls "C:\Data" /grant Everyone:(OI)(CI)R
Deny Write to Guest	icacls "C:\Data" /deny Guest:(OI)(CI)W
Remove inheritance	icacls "C:\Data" /inheritance:r
Enable inheritance	icacls "C:\Data" /inheritance:e
Full control to Admins	icacls "C:\Data" /grant Administrators:(OI)(CI)F

Important Notes

- Use Get-Acl before making any changes.
- Be careful with Deny permissions.
- Use inheritance carefully.
- Test with non-critical files first.
- Always keep backup of important ACLs.

Quick Reference Table

Command	Purpose
Get-Acl -Path <Path>	View permissions (ACL)
Set-Acl -Path <Path> -AclObject <ACL>	Apply ACL
icacls <Path> /grant <User>:<Rights>	Grant permissions
icacls <Path> /deny <User>:<Rights>	Deny permissions
icacls <Path> /remove <User>	Remove permissions
icacls <Path> /inheritance:r	Disable inheritance
icacls <Path> /t	Apply to subfolders

Permission Management Workflow

1. Check Current — Get-Acl → 2. Plan Changes — Decide Rights → 3. Apply Changes — Set-Acl /icacls → 4. Verify — Check Access → 5. Backup — Save ACL

Summary

PowerShell gives you full control over permissions.
Use Get-Acl, Set-Acl & icacls to manage access easily.
Secure your files, control access & protect your data!

Secure. Control. Protect. | Get-Acl • Set-Acl • icacls /grant • icacls /deny • Backup ACLs | PowerShell Secures Your Permissions!

ARCHIVES in PowerShell

Compress & Extract Files and Folders Easily!

ZIP • Compress • Extract • Archive • Files • Folders

ZIP Format | Compress Files | Extract Archive | Files & Folders | Archive Management | Save Space | Fast & Easy | Info Details

Why Use Archive Cmdlets?

- ✓ Compress files to save disk space
- ✓ Create backups of important data
- ✓ Easy to share files and folders
- ✓ Extract archives in one simple command
- ✓ Built-in cmdlets, no extra tools needed!

Syntax

```
Compress-Archive
Expand-Archive
(Optional Parameters)
-Path, -DestinationPath, -Force
```

Key Points

- Compress-Archive creates a .zip file.
- Expand-Archive extracts .zip files.
- Works with files and folders.
- Use -Force to overwrite existing files.
- Only supports ZIP format natively.

Common Parameters

Parameter	Description
-Path	Files or folders to compress/extract
-DestinationPath	Destination ZIP file or extract location
-Force	Overwrite existing files without prompt
-LiteralPath	Uses literal path, avoids wildcard issues
-Update	(Compress-Archive) Update existing ZIP

Pro Tip

- Use ZIP for maximum compatibility.
- Combine with wildcard (*) for bulk operations.
- Always use -Force in scripts for automation.
- Use Expand-Archive for safe extraction.

Note

- Only ZIP format is supported by default.
- Large archives may take time to process.
- Existing files may be overwritten without -Force.
- Ensure enough disk space before compressing.

1 Compress-Archive (Create ZIP)

Create a ZIP archive from files or folders.

```
Compress-Archive -Path "C:\Data\MyFolder" -DestinationPath "C:\Backup\MyFolder.zip"
```

Output (Sample)

Property	Value
Path	C:\Backup\MyFolder.zip
Length	1,245,678 bytes
LastWriteTime	25-05-2024 10:30:15
Compression	Optimal
Status	Success

Use when:

- You want to create a ZIP archive.
- You want to backup files or folders.
- You need to share multiple files easily.

2 Expand-Archive (Extract ZIP)

Extract files or folders from a ZIP archive.

```
Expand-Archive -Path "C:\Backup\MyFolder.zip" -DestinationPath "C:\Extracted\MyFolder"
```

Output (Sample)

Property	Value
ArchivePath	C:\Backup\MyFolder.zip
DestinationPath	C:\Extracted\MyFolder
FilesExtracted	125 files
FoldersExtracted	18 folders
Status	Success

Use when:

- You want to extract a ZIP archive.
- You want to restore files from a backup.
- You want to access compressed files.

Compress with Wildcard

Compress all .log files in a folder.

```
Compress-Archive -Path "C:\Logs\*.log" -DestinationPath "C:\Backup\Logs.zip"
```

Use when:

- You want to compress multiple files.
- You want to use patterns for selection.

Extract Specific Folder

Extract ZIP to a specific location.

```
Expand-Archive -Path "C:\Backup\Data.zip" -DestinationPath "D:\Data\Extracted"
```

Use when:

- You want to extract to a custom folder.
- You want to organize extracted data.

Extract with Overwrite

Extract and overwrite existing files.

```
Expand-Archive -Path "C:\Backup\Data.zip" -DestinationPath "C:\Data" -Force
```

Use when:

- You want to overwrite existing files.
- You want to automate extraction.

Quick Reference Table

Command	Purpose
Compress-Archive	Create ZIP archive
Expand-Archive	Extract ZIP archive
-Path	Specify files or folders
-DestinationPath	Set output path
-Force	Overwrite existing files
-LiteralPath	Use literal path
-Update	Update existing ZIP

Common Examples

Compress a Folder

```
Compress-Archive -Path "C:\Data\Project" -DestinationPath "C:\Backup\Project.zip"
```

Compress Multiple Files

```
Compress-Archive -Path "C:\Data\*.txt" -DestinationPath "C:\Backup\TextFiles.zip"
```

Extract ZIP Archive

```
Expand-Archive -Path "C:\Backup\Project.zip" -DestinationPath "C:\Extracted\"
```

Extract and Overwrite

```
Expand-Archive -Path "C:\Backup\Project.zip" -DestinationPath "C:\Extracted\" -Force
```

Update Existing ZIP

```
Compress-Archive -Path "C:\Data\NewFiles" -DestinationPath "C:\Backup\Project.zip" -Update
```

Archive Workflow

1. Select Files/Folders — Choose files or folders → 2. Compress — Use Compress-Archive → 3. Archive Created — ZIP file is ready → 4. Extract — Use Expand-Archive → 5. Access Files — Files are ready to use

Archive Benefits

Save Disk Space | Safe Backup | Easy Sharing | Better Organization

Summary

PowerShell archive cmdlets make compression and extraction easy. Use Compress-Archive to create ZIP files and Expand-Archive to extract them. Perfect for backup, sharing & organization!

Compress. Extract. Backup. | Compress-Archive • Expand-Archive | PowerShell Makes Archive Management Easy!

WEB REQUESTS in PowerShell

Send HTTP Requests & Work with Web Data Easily!

Invoke-WebRequest • Invoke-RestMethod • GET • POST • API • JSON • HTTP

HTTP HTTPS | GET | POST | Headers | Parameters | JSON | Response | Status Code | API | Info

Why Use Web Requests?

- ✓ Access web pages & download content
- ✓ Consume REST APIs & web services
- ✓ Send and receive JSON data
- ✓ Automate data collection & integration
- ✓ Perfect for scripts, APIs & automation

Syntax

```
Invoke-WebRequest -Uri <URL> [Options]
Invoke-RestMethod -Uri <URL> [Options]
```

Key Points

- Invoke-WebRequest gets HTML, files, images.
- Invoke-RestMethod works best with APIs (JSON/XML).
- GET is used to retrieve data.
- POST is used to send data to a server.
- Use -Headers to add custom headers.
- Use -Body to send data in POST requests.
- Check StatusCode for success or error.

Common Parameters

Parameter	Description
-Uri	The URL to send request to
-Method	GET, POST, PUT, DELETE, etc.
-Headers	Add custom headers
-Body	Data to send in POST/PUT
-OutFile	Save response to a file
-UseBasicParsing	Use basic parsing (faster)
-ContentType	Set content type (e.g. application/json)

Pro Tip

- Use Invoke-RestMethod for APIs (JSON response).
- Use Invoke-WebRequest for webpages & files.
- Always check StatusCode and handle errors.
- Use -OutFile to download large files.

Note

- Some sites may block requests (403 Forbidden).
- Always respect website terms & API limits.
- Use -Headers for authentication if required.
- For large data, handle pagination if needed.

1 Invoke-WebRequest (Web Pages & Files)

Gets content from a website, returns HTML, files, images, etc.

```
Invoke-WebRequest -Uri "https://example.com"
Invoke-WebRequest -Uri "https://example.com/page" -OutFile "page.html"
```

Output (Sample)

Property	Description
StatusCode	200
StatusDescription	OK
Content	HTML content of the page
RawContent	Raw bytes of the response
Links	All links found in the page
Images	All images found in the page

Use when:

- You want to download a webpage.
- You want to download files, images, etc.

2 Invoke-RestMethod (APIs & JSON)

Sends request to REST API and parses JSON/XML response.

```
Invoke-RestMethod -Uri "https://api.example.com/data"
Invoke-RestMethod -Uri "https://api.example.com/users" -Method Get
```

Output (Sample - JSON)

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "status": "active"
}
```

Use when:

- You are working with APIs.
- You want JSON/XML data as objects.

3 GET Request (Retrieve Data)

Use GET method to get data from server.

```
Invoke-RestMethod -Uri "https://api.example.com/dters"
-Method Get
```

Output (Sample)

Id	Name	Email	Status
1	John	john@example.com	active
2	Alice	alice@example.com	active
3	Bob	bob@example.com	inactive

Use when:

- You want to retrieve data.
- GET is default method.

4 POST Request (Send Data)

Use POST method to send data to server.

```
$body = @{ name="John"; age=25 } | ConvertTo-Json
Invoke-RestMethod -Uri "https://api.example.com/users" -Method Post
-Body $body -ContentType "application/json"
```

Output (Sample)

```
{
  "success": true,
  "message": "User created successfully",
  "id": 101
}
```

Use when:

- You want to send or create data.
- Common in APIs for create/update operations.

Headers (Add Custom Headers)

```
$headers = @{
  "Authorication" = "Bearer TOKEN123"
  "Accept" = "application/json"
}
Invoke-RestMethod -Uri "https://api.example.com/data"
-Headers $headers
```

Use when:

- API needs authentication.
- You need to send custom headers.

Handle Parameters (Query String)

```
$params = @{
  page = 1
  limit = 10
  search = "PowerShell"
}
Invoke-RestMethod -Uri "https://api.example.com/s-
earch" -Method Get -Body $params
```

Use when:

- You need to send query parameters.
- Useful for search, filter, pagination.

Check Status Code

```
$response = Invoke-WebRequest -Uri "https://example.com"
$response.StatusCode
$response.StatusDescription
```

Output (Sample)

StatusCode	StatusDescription
200	OK

Use when:

- You want to verify request result.
- Handle success or error codes.

Save Response to File

```
Invoke-WebRequest -Uri "https://example.com/file.pdf"
-OutFile "C:\Downloads\file.pdf"
```

Use when:

- You want to download files.
- Save images, docs, PDFs etc.

Error Handling (Try..Catch)

```
try {
  Invoke-RestMethod -Uri "https://api.example.com/data"
}
catch {
  Write-Host "Error: $_" -ForegroundColor Red
}
```

Use when:

- You want to handle errors.
- Avoid script breaks.

Common HTTP Status Codes

Code	Meaning	Description
200	OK	Request successful
201	Created	Resource created successfully
400	Bad Request	Invalid request
401	Unauthorized	Authentication required
403	Forbidden	Access denied
404	Not Found	Resource not found
500	Internal Server Error	Server error

Compare: WebRequest vs RestMethod

Feature	Invoke-WebRequest	Invoke-RestMethod
Best For	Web pages & files	APIs & JSON/XML
Returns	HTML, files, images	Objects (JSON/XML)
Parsing	Parses HTML content	Automatically parses JSON/XML
Speed	Slower for APIs	Faster for APIs
Use Case	Download content	Work with API data

Real-World Examples

Get public IP

```
Invoke-RestMethod -Uri "https://api.ipify.org?format=json"
```

Get current weather

```
Invoke-RestMethod -Uri "https://api.openweathermap.org/data/2.5/weather" -Method Get -Body @{ [illegible]
[illegible] }
```

Get latest GitHub repos

```
Invoke-RestMethod -Uri "https://api.github.com/users/microsoft/repos" -Method Get
```

Download file

```
Invoke-WebRequest -Uri "https://example.com/data.zip" -OutFile "data.zip"
```

Quick Reference Table

Task	Command Example
Get webpage	Invoke-WebRequest -Uri "https://example.com"
Get API data (GET)	Invoke-RestMethod -Uri "https://api.com/data" -Method Get
Send data (POST)	Invoke-RestMethod -Uri "https://api.com/data" -Method Post -Body $body -ContnttType "application/json"
Add headers	Invoke-RestMethod -Uri "https://api.com" -Headers @{Authorication="Beroer TOKEN"}
Save to file	Invoke-WebRequest -Uri "https://file.com/file.zip" -OutFile "file.zip"
Check status code	(Invoke-WebRequest -Uri "https://example.com").StatusCode

Web Request Workflow

1. Set URL — Choose the API or webpage
→ 2. Choose Method — GET / POST / PUT / DELETE
→ 3. Add Headers & Params — Set headers, query or body
→ 4. Send Request — Invoke-WebRequest / RestMethod
→ 5. Get Response — JSON, XML or Content
→ 6. Process & Use — Work with data

Summary

PowerShell makes web requests easy and powerful.
Use Invoke-WebRequest for web pages & files.
Use Invoke-RestMethod for APIs & JSON/XML.
GET for retrieving data, POST for sending data.
Automate web tasks, APIs, downloads & integrations!

Connect. Automate. Integrate. | Invoke-WebRequest • Invoke-RestMethod • GET • POST • API • HTTP | PowerShell Connects You to the Web!

REMOTE POWERSHELL

Manage Remote Computers & Run Commands Anywhere!

Remote Computer • Session • Command • WinRM • PowerShell Remoting

Remote Computer | Session Management | Run Commands | WinRM Protocol | Secure Connection | Scripts & Automation | Info & Help

Why Use Remote PowerShell?

- ✓ Manage multiple computers remotely
- ✓ Run commands & scripts on remote systems
- ✓ Automate admin tasks at scale
- ✓ Use WinRM for secure communication
- ✓ Perfect for IT admins & automation

Syntax

```
Enter-PSSession -ComputerName <Computer>
Invoke-Command -ComputerName <Computer>
New-PSSession -ComputerName <Computer>
Copy-Item -ToSession / -FromSession
```

Key Points

- PowerShell Remoting uses WinRM (Port 5985/5986).
- Enable WinRM before creating remote sessions.
- Use Enter-PSSession for interactive session.
- Use Invoke-Command for one-time commands.
- Use New-PSSession to create reusable sessions.
- Supports Credential, Scripts, Files & Modules.

Common Parameters

Parameter	Description
-ComputerName	Target remote computer name or IP
-Credential	Specify user credentials
-ScriptBlock	Script or command to execute
-ArgumentList	Arguments for the command/script
-Session	Use an existing PSSession
-Authentication	Authentication method (Default, Negotiate)
-Port	WinRM port (5985 HTTP, 5986 HTTPS)
-UseSSL	Use HTTPS for encrypted connection

WinRM Ports

Protocol	Port	Use
HTTP	5985	Unencrypted (Default)
HTTPS	5986	Encrypted (Secure)

Note

- Ensure WinRM is enabled on remote computer.
- Use firewall rules to allow WinRM traffic.
- Use HTTPS (5986) for secure connections.
- You must have admin rights on remote system.

1 Enter-PSSession (Interactive Session)

Start an interactive PowerShell session on remote computer.

```
Enter-PSSession -ComputerName Server01
```

Example (Output)

```
[Server01]: PS C:\Users\Admin> Get-Process
Handles  NPM(K)  PM(K)  CPU(s)  Id   ProcessName
123      20      15000  0.23    456  powershell
234      25      18000  0.31    789  svchost
345      15      12000  0.12    101  system
```

Use when:

- You want a full interactive remote session.
- You want to run multiple commands interactively.

2 Invoke-Command (Run Command)

Run a command or script on one or more remote computers.

```
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service }
```

Example (Output)

```
Status   Name     DisplayName
Running  WinRM    Windows Remote Management
Running  Spooler  Print Spooler
Stopped  WSearch  Windows Search
```

Use when:

- You want to run command/script remotely.
- You don't need interactive session.

3 New-PSSession (Create Session)

Create a persistent session to reuse for multiple commands.

```
$session = New-PSSession -ComputerName Server01
```

Example (Output)

```
Id  Name      ComputerName  State   ConfigurationName
--  --------  ------------  ------  -----------------
1   Session1  Server01      Opened  Microsoft.PowerShell
```

Use when:

- You want to create reusable sessions.
- You want better performance for multiple calls.

4 Using Existing Session

Run command using an existing PSSession.

```
Invoke-Command -Session $session -ScriptBlock { Get-ComputerInfo }
```

Example (Output)

```
WindowsProductName            : Windows Server 2019 Standard
WindowsVersion                : 10.0.17763
OsHardwareAbstractionLayer    : 10.0.17763.418
CsDomain                      : WORKGROUP
CsManufacturer                : Microsoft Corporation
```

Use when:

- You already have a session and want to run commands.
- Faster execution for multiple operations.

Copy Files to Remote Computer

Copy local file to remote session.

```
Copy-Item -Path "C:\Install\setup.exe" -ToSession $session -Destination "C:\Temp\"
```

Use when:

- You want to copy files to remote computer.
- Useful for software deployment.

Copy Files from Remote Computer

Copy file from remote session to local.

```
Copy-Item -Path "C:\Temp\log.txt" -FromSession $session -Destination "C:\Backup\"
```

Use when:

- You want to download files from remote system.
- Useful for logs & reports.

Disconnect & Remove Session

Close or remove a PowerShell session.

```
Remove-PSSession -Session $session
```

Use when:

- You want to close sessions after work.
- Helps free system resources.

Enable WinRM (On Remote Computer)

Enable WinRM to allow remote connections.

```
Enable-PSRemoting -Force
```

Use when:

- WinRM is not enabled on remote system.
- First time setup for remoting.

Test WinRM Connection

Check if remote computer is reachable via WinRM.

```
Test-WSMan -ComputerName Server01
```

Example (Output)

```
wsmid            : http://server01:5985/wsman/id/1
ProductVersion   : OS: 10.0.17763 SP: 0.0
WSManStackVersion : 3.0
```

Use when:

- You want to test WinRM connectivity.
- Troubleshooting connection issues.

Authentication Options

Method	
Default	Uses default authentication (Kerberos/NTLM)
Basic	Basic authentication (Use HTTPS)
Negotiate	Negotiate (Kerberos or NTLM)
CredSSP	Delegates credentials to remote computer
Certificate	Uses client certificate for authentication

Common Examples

Task	Command
Get processes on remote PC	Invoke-Command -ComputerName PC01 -ScriptBlock { Get-Process }
Get service list	Invoke-Command -ComputerName PC01 -ScriptBlock { Get-Service }
Get disk info	Invoke-Command -ComputerName PC01 -ScriptBlock { Get-WmiObject Win32_LogicalDisk }
Restart remote computer	Invoke-Command -ComputerName PC01 -ScriptBlock { Restart-Computer -Force }
Run script file	Invoke-Command -ComputerName PC01 -FilePath C:\Scripts\Check.ps1

Session Information

Command	Purpose
Get-PSSession	List all active PowerShell sessions
Get-PSSession | Format-List	View detailed session info
Set-PSSessionConfiguration	Configure session settings
Get-PSSessionConfiguration	List session configurations

Security Best Practices

- Use HTTPS (5986) for encrypted communication.
- Use strong passwords & least privilege.
- Enable firewall only for trusted networks.
- Regularly update systems & PowerShell.
- Monitor & log remote access activities.

Remote PowerShell Workflow

1. Enable WinRM (Enable-PSRemoting -Force) → 2. Create Session (New-PSSession) → 3. Run Commands (Invoke-Command) → 4. Copy Files (Copy-Item -ToSession) → 5. Disconnect (Remove-PSSession)

Summary

PowerShell Remoting lets you manage remote computers easily. Use Enter-PSSession for interactive access and Invoke-Command for quick one-time commands. WinRM makes it secure & powerful!

Manage Remote Systems. Automate Tasks. Save Time. | Enter-PSSession • Invoke-Command • New-PSSession • WinRM | PowerShell Remoting – Work from Anywhere!

POWERSHELL SCRIPTS

Automate Tasks with Powerful .ps1 Scripts!

Script • Automate • Execute • Manage • Repeat

PS1 File | Script | Variables | Commands | Functions | Parameters | Automation | Execution | Info

Why Use PowerShell Scripts?

- ✓ Automate repetitive tasks
- ✓ Save time and reduce errors
- ✓ Manage systems & configurations
- ✓ Reusable and easy to maintain
- ✓ Perfect for IT admins & developers

Syntax

```
# This is a comment
$variable = "Value"
Write-Host "Hello World"
Get-Process | Where-Object {$_.CPU -gt 100}
function My-Function {
   param($Param1)
   Write-Host $Param1
}
My-Function -Param1 "Test"
```

Key Points

- Scripts use .ps1 extension.
- Variables start with $ sign.
- Use functions to organize code.
- Parameters make scripts flexible.
- Run scripts with: .\script.ps1
- Execution policy may block scripts.

Common Parameters

Parameter	Description
-File <path>	Run a script file.
-ExecutionPolicy	Set execution policy.
-NoProfile	Run without loading profile.
-WindowStyle	Set window style.
-Verbose	Show detailed output.
-WhatIf	Show what would happen.

Pro Tip

- Always test scripts in a safe environment.
- Use -WhatIf to preview changes.
- Write comments for better readability.
- Use functions to reuse code.

Note

- Scripts may be blocked by Execution Policy.
- Run as Administrator if script needs admin rights.
- Always backup important data before automation.
- Use try/catch for error handling.

1 Variables in Scripts

Store data using variables.

```
$name = "CodeSardar"
$age = 32
Write-Host "Name: $name"
Write-Host "Age: $age"
```

Output (Sample)

```
Name: CodeSardar
Age: 32
```

Use when:

- You need to store values.
- You want to use data multiple times.

2 Functions in Scripts

Create reusable blocks of code.

```
function Greet-User {
   param($UserName)
   Write-Host "Hello, $UserName!"
}
Greet-User -UserName "CodeSardar"
```

Output (Sample)

```
Hello, CodeSardar!
```

Use when:

- You want to reuse code.
- You want to keep script organized.

3 Parameters in Scripts

Make scripts dynamic with parameters.

```
param(
   [string]$Name,
   [int]$Age
)
Write-Host "Name: $Name, Age: $Age"
```

Output (Sample)

```
Name: CodeSardar, Age: 32
```

Use when:

- You want to pass values while running.
- You want flexible scripts.

4 Conditional Statements

Run code based on conditions.

```
$disk = (Get-WmiObject Win32_LogicalDisk
  -Filter "DriveType=3").Size/1GB
if ($disk -gt 100) {
   Write-Host "Disk size is greater than 100 GB"
} else {
   Write-Host "Disk size is 100 GB or less"
}
```

Use when:

- You want to run different code.
- You need decision making in scripts.

5 Loop in Scripts

Repeat tasks using loops.

```
for ($i=1; $i -le 5; $i++) {
   Write-Host "Count: $i"
}
```

Output (Sample)

```
Count: 1
Count: 2
Count: 3
Count: 3
Count: 4
Count: 5
```

Use when:

- You want to repeat tasks.
- You want to process multiple items.

6 Error Handling

Handle errors gracefully.

```
try {
   Get-Item "C:\NotExist.txt" -ErrorAction Stop
} catch {
   Write-Host "Error: $_"
}
```

Use when:

- You want to handle errors.
- You want script to continue.

7 Reading Input

Take input from user.

```
$name = Read-Host "Enter your name"
Write-Host "Hello, $name"
```

Output (Sample)

```
Enter your name: CodeSardar
Hello, CodeSardar!
```

Use when:

- You want user interaction.
- You want dynamic input.

8 Working with Files

Read and write files.

```
$content = Get-Content "C:\Data.txt"
$content | ForEach-Object {
   Write-Host $_
}
```

Use when:

- You want to read file content.
- You want to process files.

9 Logging in Scripts

Write logs for tracking.

```
$log = "C:\ScriptLog.txt"
Add-Content -Path $log -Value "Script started"
Get-Date | Add-Content -Path $log
Add-Content -Path $log -Value "Script finished"
```

Use when:

- You want to track script execution.
- You want logs for troubleshooting.

10 Running Scripts

Execute scripts safely.

```
# Check Execution Policy
Get-ExecutionPolicy

# Run script
.\MyScript.ps1 -Name "CodeSardar" -Age 32
```

Use when:

- You want to run your scripts.
- You want to pass parameters.

Script Structure Example

```
# Sample PowerShell Script
param(
   [string]$Name = "Guest",
   [int]$Age = 0
)
# Function
function Show-Info {
   param($Name, $Age)
   Write-Host "Name: $Name"
   Write-Host "Age: $Age"
}
# Main Script
Write-Host "Script Started..."
Show-Info -Name $Name -Age $Age
Write-Host "Script Finished."
```

← Header / Comment
← Parameters Section
← Function Definition
← Main Script Execution

Common Commands in Scripts

Command	Purpose
Write-Host	Display output to console
Get-Process	Get running processes
Get-Service	Get system services
Get-Item	Get file or folder info
Set-Item	Set file or folder info
Remove-Item	Delete file or folder
Copy-Item	Copy file or folder
Move-Item	Move or rename file/folder
Start-Process	Start a process/application
Stop-Process	Stop a running process

Execution Policy

Policy	Description
Restricted	No scripts allowed
AllSigned	Only signed scripts allowed
RemoteSigned	Local scripts allowed, remote signed only
Unrestricted	All scripts allowed (not recommended)
Bypass	No restrictions (use carefully)

Set Execution Policy

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Run as Administrator if needed.

Best Practices

- Use meaningful variable names.
- Write comments in your scripts.
- Use functions for reusable code.
- Handle errors using try/catch.
- Use logging for important scripts.
- Test scripts before production.

Security Tips

- Use least privilege.
- Do not hardcode passwords.
- Use secure credentials.
- Keep scripts in secure locations.
- Review scripts regularly.

1. Create Script — Write your code → 2. Save Script — Save with .ps1 → 3. Set Policy — Allow script execution → 4. Run Script — Execute the script → 5. Review Output — Check results & logs → 6. Automate — Schedule or reuse

Summary

PowerShell scripts make automation easy and powerful.
Use variables, functions, and parameters to build smart scripts.
Test, log, and secure your scripts for best results!

Write. Automate. Simplify. | .ps1 • Script • Automate • Execute • Manage | PowerShell Scripts – Automate Everything!

CONDITIONS in PowerShell

Make Smart Decisions in Your Scripts!

if • elseif • else • switch • -eq • -ne • -gt • -lt

if	elseif	else	switch	-eq	-ne	-gt	-lt	Info
Condition	Else If	Else	Multiple Choice	Equal	Not Equal	Greater Than	Less Than	Details

Why Use Conditions?

- ✓ Make decisions in your scripts
- ✓ Control the flow of execution
- ✓ Handle different situations
- ✓ Write smarter & dynamic scripts
- ✓ Essential for automation & logic

Syntax

```
if (Condition) {
   # Code to execute
}
elseif (Condition) {
   # Code to execute
}
else {
   # Code to execute
}

switch (Expression) {
   Case { # Code }
   Default { # Code }
}
```

Key Points

- Conditions return True or False.
- Use () to evaluate conditions.
- Use -eq, -ne, -gt, -lt for comparisons.
- Use elseif for additional conditions.
- Use switch for multiple possible values.
- PowerShell is case-insensitive.

Common Operators

Operator	Description
-eq	Equal
-ne	Not Equal
-gt	Greater Than
-lt	Less Than
-ge	Greater Than or Equal
-le	Less Than or Equal
-like	Like (Wildcard Match)
-notlike	Not Like (Wildcard Match)
-match	Regular Expression Match
-notmatch	Not Regular Expression Match

Pro Tip

- Use meaningful variable names.
- Test conditions with Write-Host.
- Use switch for many possible values.
- Keep conditions simple and readable.

1 if Statement

Executes code if condition is True.

```
$age = 18

if ($age -ge 18) {
  Write-Host "You are an adult."
}
```

Output (Sample)

```
You are an adult.
```

Use when:

- You want to execute code if condition is True.
- Best for single condition.

2 if...elseif...else Statement

Checks multiple conditions.

```
$score = 75
if ($score -ge 90) {
  Write-Host "Grade: A"
} elseif ($score -ge 75) {
  Write-Host "Grade: B"
} else {
  Write-Host "Grade: C"
}
```

Output (Sample)

```
Grade: B
```

Use when:

- You have multiple conditions.
- Provides different outcomes.

3 else Statement

Executes code when condition is False.

```
$isActive = $false

if ($isActive) {
  Write-Host "Active"
} else {
  Write-Host "Not Active"
}
```

Output (Sample)

```
Not Active
```

Use when:

- Execute code when condition is False.
- Default block for if.

4 Nested if Statement

Use if inside another if.

```
$age = 20
$hasLicense = $true

if ($age -ge 18) {
  if ($hasLicense) {
    Write-Host "You can drive."
  } else {
    Write-Host "Get a license first."
}
```

Output (Sample)

```
You can drive.
```

Use when:

- You need to check multiple levels.
- Useful for detailed logic.

5 switch Statement

Checks multiple possible values.

```
$day = "Monday"

swtich ($day) {
   "Monday"    { Write-Host "Start of week" }
   "Friday"    { Write-Host "Almost weekend" }
   "Sunday"    { Write-Host "Weekend!" }
   Default     { Write-Host "Mid week" }
}
```

Output (Sample)

```
Start of week
```

Use when:

- You have many possible values.
- Cleaner and easier to read.

6 Comparison Operators

Compare numeric values.

Operator	Example	Result
-eq	5 -eq 5	True
-ne	5 -ne 3	True
-gt	10 -gt 5	True
-lt	3 -lt 7	True
-ge	5 -ge 5	True
-le	2 -le 4	True

7 String Comparison

Compare string values.

Operator	Example	Result
-eq	"abc" -eq "abc"	True
-ne	"abc" -ne "def"	True
-like	"a*" -like "abc"	True
-notlike	"b*" -notlike "abc"	True
-match	"a.*c" -match "abc"	True
-notmatch	"d.*" -notmatch "abc"	True

8 Logical Operators

Combine multiple conditions.

Operator	Example	Result
-and	($a -gt 5) -and ($b -lt 10)	True
-or	($a -gt 5) -or ($b -lt 10)	True
-not	-not ($a -eq 5)	True
!	!($a -eq 5)	True

9 Example: Complex Condition

Combine multiple operators.

```
$age = 25
$salary = 50000
if ($age -ge 18 -and $salary -gt 30000) {
  Write-Host "Eligible for loan."
} else {
  Write-Host "Not eligible."
}
```

Output (Sample)

```
Eligible for loan.
```

10 Example: switch with Numbers

Switch with numeric values.

```
$choice = 2
swtich ($choice) {
   1  { Write-Host "One" }
   2  { Write-Host "Two" }
   2  { Write-Host "Three" }
   Default  { Write-Host "Other" }
}
```

Output (Sample)

```
Two
```

Common Use Cases

- ✓ Validate user input
- ✓ Check file or folder existence.
- ✓ Compare values and make decisions
- ✓ Process data based on conditions
- ✓ Control script flow

Truthy and Falsy Values

True (Truthy)	False (Falsy)
$true	$false
1, -1, 100	0
"Yes", "True", "OK"	"".
Non-empty strings	@() (empty array)
@(), @(1), @("a")	$null
Any non-null object	$false

Tips for Better Conditions

- Always use () for clarity.
- Use meaningful variable names.
- Avoid complex one-liners.
- Use switch for many possible values.
- Test your conditions thoroughly.

Common Mistakes

- ✗ Forgetting () in conditions.
- ✗ Using = instead of -eq.
- ✗ Ignoring case-insensitive behavior.
- ✗ Not handling else case.
- ✗ Overusing nested if statements.

Quick Reference

Syntax	Use
if (condition) { }	Single condition
if ( ) elseif ( ) else ( )	Multiple conditions
switch (value) { }	Multiple possible values
-eq, -ne, -gt, -lt	Compare values
-and, -or, -not	Combine conditions

Condition Workflow

1. Set Variables — Get input or data →
2. Check Condition — Evaluate the condition →
3. Execute Code — Run appropriate block →
4. Get Result — Output or action →
5. Continue Script — Move to next step

Summary

Conditions help your scripts make smart decisions.
Use if, elseif, else for simple logic and switch for multiple choices.
Combine operators for advanced checks and automation!

Think. Check. Decide. | if • elseif • else • switch • -eq • -ne • -gt • -lt | Smart Scripts. Better Automation.

LOOPS in PowerShell

Automate Repetitive Tasks with Powerful Loops! 🚀

for • foreach • while • do • until • break • continue • Loop

Loop	for	foreach	while	do	until	break	continue	Info
Repeat	Count Loop	For Each Item	While True	Do First	Until True	Exit Loop	Next Iteration	Details

Why Use Loops?

- ✓ Automate repetitive tasks
- ✓ Save time and reduce errors
- ✓ Process large data sets easily
- ✓ Improve efficiency and productivity
- ✓ Essential for scripting and automation

Syntax

```
for (...) {
    # code block
}

foreach ($item in $collection) {
    # code block
}
while (Condition) {
    # code block
}
do {
    # code block
} while (Condition)

do {
    # code block
} until (Condition)
```

Key Points

- Loops repeat code until a condition is met.
- Use break to exit the loop early.
- Use continue to skip to next iteration.
- Choose the right loop for the job.
- Infinite loops can be dangerous.
- Test your conditions carefully.

Common Keywords

Keyword	Use
for	Loop a specific number of times
foreach	Loop through each item in a collection
while	Loop while condition is True
do	Execute code block first
until	Loop until condition is True
break	Exit the loop
continue	Skip current iteration, continue next

Pro Tip

- Always test your loops with small data first.
- Use Write-Host to see what's happening.
- Avoid infinite loops: ensure condition changes.
- Use break and continue wisely.

1 for Loop

Repeat code a specific number of times.

```
for ($i = 1; $i -le 5; $i++) {
  Write-Host "Count: $i"
}
```

Output (Sample)

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Use when:

- You know how many times to loop.
- Perfect for counters and indexes.

2 foreach Loop

Loop through each item in a collection.

```
$names = @("John","Alice","Bob")
foreach ($name in $names) {
    Write-Host "Hello, $name!"
}
```

Output (Sample)

```
Hello, John!
Hello, Alice!
Hello, Bob!
```

Use when:

- You want to process each item.
- Great for arrays, lists, or objects.

3 while Loop

Repeat while the condition is True.

```
$i = 1
while ($i -le 3) {
    Write-Host "Number: $i"
    $i++
}
```

Output (Sample)

```
Number: 1
Number: 2
Number: 3
```

Use when:

- You don't know how many times.
- Condition controls the loop.

4 do...while Loop

Execute code block first, then check condition.

```
$i = 1
do {
    Write-Host "Run: $i"
    $i++
} while ($i -le 3)
```

Output (Sample)

```
Run: 1
Run: 2
Run: 3
```

Use when:

- You need to run code at least once.
- Good for menus and user input.

Quick Reference Table

Loop / Keyword	Purpose
for	Loop a fixed number of times
foreach	Loop through collection items
while	Loop while condition is True
do...while	Run code first, then check
do...until	Loop until condition is True
break	Exit loop immediately
continue	Skip to next iteration

5 do...until Loop

Repeat code until the condition is True.

```
$i = 1
do {
   Write-Host "Try: $i"
   $i++
} until ($i -gt 3)
```

Output (Sample)

```
Try: 1
Try: 2
Try: 3
```

Use when:

- You want to repeat until success.
- Condition decides when to stop.

6 break Statement

Exit the loop before it completes.

```
for ($i = 1; $i -le 10; $i++) {
   if ($i -eq 5) { break }
   Write-Host "Number: $i"
}
```

Output (Sample)

```
Number: 1
Number: 2
Number: 3
Number: 4
```

Use when:

- Yop loop early when condition met.
- Useful for error handling.

7 continue Statement

Skip the current iteration, continue next.

```
for ($i = 1; $i -le 5; $i++) {
   if ($i -eq 3) { continue }
   Write-Host "Number: $i"
}
```

Output (Sample)

```
Number: 1
Number: 2
Number: 4
Number: 5
```

Use when:

- Skip unwanted values or conditions.
- Keeps the loop running.

8 while True (Infinite Loop)

Run forever until stopped manually.

```
while ($true) {
  Write-Host "Press Ctrl+C to stop..."
  Start-Sleep -Seconds 1
}
```

Use when:

- For continuous monitoring.
- Remember to stop it! (Ctrl+C)

Common Use Cases

- Process files in a folder
- Read user input until valid
- Retry operations until success
- Monitor logs continuously
- Generate reports or data

Note

- Be careful with infinite loops.
- Always ensure the condition will change.
- Use break or Ctrl+C to stop loops.
- Test thoroughly before production use.

Best Practices

- ✓ Use meaningful variable names.
- ✓ Keep loop logic simple.
- ✓ Add comments for complex loops.
- ✓ Use Write-Host for debugging.
- ✓ Validate data before looping.

Loop Workflow

1. Initialize — Set variables / counters → 2. Check Condition — Evaluate the condition → 3. Execute Code — Run the code block → 4. Update — Change variables / counters → 5. Repeat — Loop until condition fails

Example: Sum of Numbers 1 to 5 (for Loop)

```
$sum = 0
for ($i = 1; $i -le 5; $i++) {
    $sum += $i
}
Write-Host "Total Sum: $sum"
```

Output:

```
Total Sum: 15
```

Summary

Loops are the heart of automation in PowerShell. Choose the right loop, write clean conditions, and control the flow with break and continue. Master loops, master PowerShell! 💪

Automate. Repeat. Achieve More. 🎯 | for • foreach • while • do • until • break • continue | PowerShell Loops – Work Smarter! 🚀

FUNCTIONS in PowerShell

Write Reusable Code with PowerShell Functions! 🚀

function • param() • return • Parameters • Arguments • Output • Reusable Code • Script

function	param()	return	Parameters	Arguments	Output	Reusable Code	Script
Define Function	Parameters	Return Value	Input Definition	Values Passed	Result	Use Again	.ps1 File

Why Use Functions?

- ✔ Organize and structure your code
- ✔ Reuse code and avoid repetition
- ✔ Make scripts easy to read & maintain
- ✔ Accept parameters for flexibility
- ✔ Return results for further processing

Syntax

```
function FunctionName {
    param(
      [Parameter(Mandatory=$true)]
      [string]$Name,
      [int]$Age
    )
    # Function body
    Write-Host "Hello, $Name!"
    return $Age
}
```

Key Points

- Functions start with the 'function' keyword.
- Use param() to define parameters.
- Parameters act as placeholders for values.
- Use return to send back a value.
- Functions improve code reusability.
- Call functions by name with arguments.

Common Parameters

Parameter	Description
Mandatory	Makes parameter required
Position	Defines the order (default 0,1,2...)
DefaultValue	Sets a default value
ValidateSet	Restrict values to a list
ValidateNotNull	Ensures value is not null
ValueFromPipeline	Accept input from pipeline

Pro Tip

- Use meaningful function names.
- Keep functions short and focused.
- Add comment-based help.
- Test functions with different inputs.

1 Simple Function (No Parameters)

Create and run a basic function.

```
function Say-Hello {
  Write-Host "Hello, PowerShell!"
}
Say-Hello
```

Output (Sample)

```
Hello, PowerShell!
```

Use when:

- You need a simple reusable task.
- No input is required.

2 Function with Parameters

Accept input using parameters.

```
function Greet-User {
  param($Name)
  Write-Host "Hello, $Name!"
}
Greet-User -Name "Jeet"
```

Output (Sample)

```
Hello, Jeet!
```

Use when:

- You need to accept user input.
- Make function dynamic.

3 Multiple Parameters

Use multiple parameters in a function.

```
function Add-Numbers {
  param($a, $b)
  $sum = $a + $b
  return $sum
}
Add-Numbers -a 10 -b 20
```

Output (Sample)

```
30
```

Use when:

- You need multiple inputs.
- Perform operations and return result.

4 Default Parameter Values

Set default values for parameters.

```
function Greet-User {
  param($Name = "Guest")
  Write-Host "Hello, $Name!"
}
Greet-User     # Uses default value
```

Output (Sample)

```
Hello, Guest!
```

Use when:

- Provide default values.
- Make parameters optional.

5 Returning Values

Return values from a function.

```
function Get-Square {
  param($num)
  return $num * $num
}
Get-Square -num 5
```

Output (Sample)

```
25
```

Use when:

- You need to use result elsewhere.
- Chain functions or store output.

6 Advanced Parameters

Use validation, position & mandatory.

```
function New-User {
  param(
    [Parameter(Mandatory=$true, Position=0)]
    [string]$UserName,
    [int]$Age = 18,
    [ValidateSet("M","F")] [string]$Gender
  )
  Write-Host "User: $UserName, Age: $Age,
   Gender: $Gender"
}
New-User -UserName "Jeet" -Gender "M"
```

Output (Sample)

```
User: Jeet, Age: 18, Gender: M
```

Use when:

- Enforce input rules.
- Make functions robust and user-friendly.

7 Pipeline Input

Accept input from the pipeline.

```
function Get-FileInfo {
  param(
    [Parameter(ValueFromPipeline=$true)]
    [string]$Path
  )
  process {
    Get-Item $Path | Select-Object Name, Length
  }
}
"C:\Temp\file1.txt", "C:\Temp\file2.txt" | Get-FileInfo
```

Output (Sample)

Name	Length
file1.txt	1200
file2.txt	3400

Use when:

- Work with pipeline input.
- Process multiple items.

8 Function in Script File

Save function in a .ps1 file and use it.

```
# MyFunctions.ps1
function Get-DateTime {
  return Get-Date
}
Export-ModuleMember -Function Get-DateTime

# Using in another script
. .\MyFunctions.ps1
Get-DateTime
```

Output (Sample)

```
25-05-2025 10:30:45
```

Use when:

- Reuse functions across scripts.
- Keep code organized.

9 Function with Output Object

Return custom objects.

```
function Get-UserInfo {
  param($Name, $Age)
  return [PScustomObject]@{
    Name  = $Name
    Age   = $Age
  }
}
Get-UserInfo -Name "Jeet" -Age 29
```

Output (Sample)

Name	Age
Jeet	29

Use when:

- Return structured data.
- Easier to work with in scripts.

10 Nested Function Call

Call one function inside another.

```
function Get-Sum {
  param($a, $b)
  return $a + $b
}
function Get-DoubleSum {
  param($a, $b)
  $sum = Get-Sum -a $a -b $b
  return $sum * 2
}
Get-DoubleSum -a 5 -b 10
```

Output (Sample)

```
30
```

Use when:

- Build complex logic.
- Reuse existing functions.

Quick Reference

Topic	Example
Define Function	function My-Func { }
Add Parameters	param($a, $b)
Mandatory Param	[Parameter(Mandatory=$true)]
Default Value	param($name = "Guest")
Return Value	return $result
Call Function	My-Func -a 1 -b 2
Pipeline Input	Get-Item | My-Func
Dot Source File	. .\MyFunctions.ps1

Common Mistakes

- ✖ Forgetting param() for inputs.
- ✖ Not using return for output.
- ✖ Overly long or complex functions.
- ✖ Ignoring input validation.
- ✖ Using global variables inside functions.

Function Workflow

1. Define Function — Use 'function' keyword
2. Add Parameters — Define inputs
3. Write Code — Add logic
4. Return Output — Use 'return'
5. Call Function — Pass arguments
6. Get Results — Use output
7. Reuse — Again & again

Example: Factorial Function

```
function Get-Factorial {
  param([int]$n)
  $fact = 1
  for ($i = 1; $i -le $n; $i++) {
    $fact *= $i
  }
  return $fact
}
Get-Factorial -n 5
```

Output: (Sample)

```
120
```

Summary

Functions make your PowerShell scripts modular, reusable and easy to maintain.

Use parameters for input, return for output, and keep your code organized!

Best Practices

- ✔ Use PascalCase for function names.
- ✔ Add comment-based help (<# ... #>).
- ✔ Keep functions short and focused.
- ✔ Validate inputs and handle errors.
- ✔ Return meaningful results.
- ✔ Test functions with different inputs.

Write Once. Use Many Times. 🎯 | function • param() • return • Parameters • Arguments • Output • Reusable Code • Script | PowerShell Functions – Code Better, Reuse Smarter! 🚀

ERROR HANDLING in PowerShell

Handle Errors Like a Pro in Your Scripts!

try • catch • finally • throw • ErrorAction • ErrorVariable • Exception • Handling

Try Block	Catch Block	finally	throw	ErrorAction	ErrorVariable	Exception	Handling
Risky Code	Handle Error	Block Block Always Runs	Throw Error Raise Exception	Control Behavior on Error	Store Error Info in Variable	Exception Object Details	Graceful Handling

Why Handle Errors?

- ✓ Prevent script crashes
- ✓ Provide meaningful feedback
- ✓ Log and troubleshoot easily
- ✓ Improve reliability and user experience
- ✓ Professional and clean scripts

Syntax

```
try {
    # Code that might throw an error
}
catch [ExceptionType] {
    # Handle the error
}
catch {
    # Handle any other error
}
finally {
    # Always runs (cleanup code)
}
```

Key Points

- Use try block for risky code.
- Use catch block to handle errors.
- Use finally block for cleanup.
- Use throw to raise custom errors.
- Control errors with -ErrorAction.
- Store errors in variables with -ErrorVariable.
- Keep scripts running even if errors occur.

Common -ErrorAction Values

Value	Description
Continue	Continue execution, show error
SilentlyContinue	Continue, hide error
Stop	Stop script, throw error (default)
Inquire	Ask user what to do
Ignore	Ignore the error completely
Suspend	Pause and debug

Pro Tip

- Use -ErrorAction Stop with try/catch.
- Use -ErrorVariable to log detailed errors.
- Always log errors for troubleshooting.
- Provide user-friendly error messages.

1 Basic try...catch

Handle simple errors.

```
try {
    Get-Item "C:\NoFile.txt" -ErrorAction Stop
}
catch {
    Write-Host "File not found!"
}
```

Output (Sample)

```
File not found!
```

Use when:

- You want to handle a specific error.
- You want to prevent script from stopping.

2 catch Specific Exception

Catch a specific type of exception.

```
try {
    [int]$x = "abc"
}
catch [System.FormatException] {
    Write-Host "Invalid number format!"
}
```

Output (Sample)

```
Invalid number format!
```

Use when:

- You want to handle a specific exception type.
- You want more control over error handling.

3 Multiple catch Blocks

Handle different exceptions differently.

```
try {
    Get-Content "C:\NoFile.txt" -ErrorAction Stop
}
catch [System.IO.FileNotFoundException] {
    Write-Host "File not found!"
}
catch [Exception] {
    Write-Host "Something else went wrong!"
}
```

Output (Sample)

```
File not found!
```

Use when:

- You want to handle multiple error types.
- You want custom messages for each error.

4 finally Block

Run code no matter what.

```
try {
    Write-Host "Doing work..."
    Get-Item "C:\NoFile.txt" -ErrorAction Stop
}
catch {
    Write-Host "Error occurred!"
}
finally {
    Write-Host "Cleanup done."
}
```

Output (Sample)

```
Doing work...
Error occurred!
Cleanup done.
```

Use when:

- You need cleanup code (close files, remove temp).
- You want code to run always.

5 Using throw

Raise a custom error.

```
function Check-Value($val) {
    if ($val -lt 1) {
        throw "Value must be 1 or greater."
    }
    return $val
}
try {
    Check-Value 0
}
catch {
    Write-Host $_.Exception.Message
}
```

Output (Sample)

```
Value must be 1 or greater.
```

Use when:

- You want to validate and stop execution.
- You want to raise custom exceptions.

6 Using -ErrorAction

Control how errors are handled.

```
Get-Item "C:\NoFile.txt" -ErrorAction SilentlyContinue

Write-Host "Script continues...."
```

Output (Sample)

```
Script continues...
```

Use when:

- You want to continue script on error.
- You want to suppress or control error output.

7 Using -ErrorVariable

Store error information in a variable.

```
Get-Item "C:\NoFile.txt" -ErrorAction
SilentlyContinue -ErrorVariable err

if ($err) {
    Write-Host "Error saved:"
    $err[0].Exception.Message
}
```

Output (Sample)

```
Error saved:
Cannot find path 'C:\NoFile.txt' because it does not exist.
```

Use when:

- You want to log or analyze errors.
- You want detailed error information.

8 Logging Errors to File

Log errors for later review.

```
try {
    Get-Content "C:\NoFile.txt" -ErrorAction Stop
}
catch {
    "$($_.Exception.Message)" | Out-File -
    FilePath "C:\error.log" -Append
    Write-Host "Error logged."
}
```

Output (Sample)

```
Error logged.
```

Use when:

- You want to keep error logs.
- You want to debug issues later.

Best Practices

- ✓ Always use -ErrorAction Stop in try block.
- ✓ Handle specific exceptions before general ones.
- ✓ Use finally for cleanup tasks.
- ✓ Log errors for debugging.
- ✓ Provide clear and user-friendly messages.
- ✓ Don't ignore errors silently.

Common Mistakes

- ✗ Not using -ErrorAction Stop in try.
- ✗ Ignoring all errors.
- ✗ Not logging errors.
- ✗ Using broad catch {} only.
- ✗ Forgetting cleanup in finally.

Pro Tips

- Use Write-Error for non-terminating errors.
- Use Try/Catch for terminating errors.
- Use $Error variable for global errors.
- Test error handling with wrong inputs.

Error Object Properties

Property	Description
Exception	Exception details
Message	Error message
CategoryInfo	Category of error
FullyQualifiedErrorId	Unique error ID
TargetObject	Object that caused error
ScriptStackTrace	Script stack trace

Example: Full Error Handling Pattern

```
try {
    # Risky code
    Get-Item "C:\NoFile.txt" -ErrorAction Stop
}
catch [System.IO.FileNotFoundException] {
    Write-Host "File not found: $($_.Exception.Message)"
}
catch [Exception] {
    Write-Host "Unexpected error: $($_.Exception.Message)"
}
finally {
    Write-Host "Finished execution."
}
```

Output (Sample)

```
File not found: Cannot find path
'C:\NoFile.txt' because it does not exist.
Finished execution.
```

Error Handling Workflow

1. Try Block — Run risky code → 2. Catch Block — Handle error → 3. Finally Block — Cleanup always → 4. Log / Store — Save error info → 5. Continue — Script continues

Handle Errors. Stay Safe. Keep Scripts Running! | try • catch • finally • throw • ErrorAction • ErrorVariable • Exception • Handling | PowerShell Error Handling – Write Better Scripts!

MODULES in PowerShell

Manage, Discover & Use Powerful PowerShell Modules! 🚀

Get-Module • Import-Module • Remove-Module • Find-Module • Install-Module

Get-Module	Import-Module	Remove-Module	Find-Module	Install-Module	Modules	Commands	Packages
List Modules	Load Module	Unload Module	Search Online	Install Package	Reusable Code	From Modules	PSGallery

Why Use Modules?

- ✓ Extend PowerShell functionality
- ✓ Use pre-built commands & tools
- ✓ Save time with reusable code
- ✓ Easy to install, update & manage
- ✓ Share and automate tasks easily

Syntax

```
# List all modules
Get-Module

# Import a module
Import-Module <ModuleName>

# Remove (unload) a module
Remove-Module <ModuleName>

# Find modules online
Find-Module <Keyword>

# Install a module
Install-Module <ModuleName>
```

Key Points

- Modules contain commands, functions & variables.
- Use Get-Module to see available modules.
- Import-Module loads module into session.
- Remove-Module unloads a module.
- Find-Module searches PowerShell Gallery.
- Install-Module installs module from PSGallery.
- Use Update-Module to keep modules updated.
- Modules make scripts clean, short & powerful.

Common Module Scopes

Scope	Description
AllUsers	Available for all users
CurrentUser	Available for current user
Process	Available in current session
LocalMachine	System-wide installation

Pro Tip

- Always install modules from PSGallery.
- Check module info before installing.
- Update modules regularly.
- Use Import-Module in scripts.
- Keep modules lightweight & focused.

1 Get-Module

List all modules in the current session.

```
Get-Module
```

Output (Sample)

ModuleType	Version	Name	ExportedCommands
Script	2.0.0.0	Microsoft.PowerShell.Utility	{Get-Date, Get-Random...}
Manifest	3.1.0	Microsoft.PowerShell.Management	{Get-Process, Stop-Process...}
Binary	1.2.0	PSReadLine	{Get-PSReadLineOption...}

Use when:

- You want to see loaded modules.
- You want to check module details.

2 Import-Module

Load a module into the current session.

```
Import-Module -Name
Microsoft.PowerShell.Utility
```

Output (Sample)

(No output if successful)

Use when:

- You want to use commands from a module.
- Load specific version or path of a module.

3 Remove-Module

Unload a module from the session.

```
Remove-Module -Name
Microsoft.PowerShell.Utility
```

Output (Sample)

(No output if successful)

Use when:

- You want to remove module from memory.
- Free resources or reload module.

4 Find-Module

Search for modules in PowerShell Gallery.

```
Find-Module -Name Az -Repository PSGallery
```

Output (Sample)

Version	Name	Repository	Description
11.5.0	Az	PSGallery	Microsoft Azure PowerShell
2.2.0	Az.Accounts	PSGallery	Microsoft Azure PowerShell...
5.0.0	Az.Storage	PSGallery	Microsoft Azure PowerShell...

Use when:

- You want to search modules online.
- Find modules by name or keyword.

5 Install-Module

Install a module from PowerShell Gallery.

```
Install-Module -Name PSWindowsUpdate
 -Repository PSGallery -Scope CurrentUser -Force
```

Output (Sample)

(No output if successful)

Use when:

- You want to install a new module.
- Install for current user or all users.

6 Import from Path

Import module from a local path.

```
Import-Module
 -Name "C:\Modules\MyModule\MyModule.psd1"
```

Output (Sample)

(No output if successful)

Use when:

- You have a module file locally.
- Testing or developing custom modules.

7 Get-Module (Details)

Show detailed information about modules.

```
Get-Module -ListAvailable
```

Output (Sample)

ModuleType	Version	Name
Script	1.0.0	MyModule
Manifest	3.0.0	PSScheduledJobs
Binary	2.1.0	ThreadJob

Use when:

- You want to see all available modules.
- Check modules not yet imported.

8 Update-Module

Update modules to the latest version.

```
Update-Module
```

Output (Sample)

ModuleName	RequiredVersion	InstalledVersion	Status
PSReadLine	2.2.6	2.2.4	Updated
Az	11.5.0	10.8.0	Updated

Use when:

- You want to keep modules up to date.
- Fix bugs and get new features.

9 Uninstall-Module

Uninstall a module.

```
Uninstall-Module
 -Name PSWindowsUpdate
```

Output (Sample)

(No output if successful)

Use when:

- You no longer need a module.
- Free disk space or remove old modules.

10 Get-Command -Module

List commands from a specific module.

```
Get-Command -Module
Name Microsoft.PowerShell.Utility
```

Output (Sample)

CommandType	Name	Version	Source
Cmdlet	Add-Member	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Clear-Variable	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Compare-Object	3.1.0.0	Microsoft.PowerShell.Utility

Use when:

- Explore commands in a module.
- Find specific commands to use.

Common Module Repositories

Repository	Description
PSGallery	Official PowerShell Gallery
NuGet	NuGet.org packages
Local	Local machine repository
Custom	Custom private repository

Popular Modules

- ✓ PSWindowsUpdate – Windows Update Management
- ✓ Az – Microsoft Azure Management
- ✓ Pester – PowerShell Testing Framework
- ✓ PSReadLine – Enhanced Console Editing
- ✓ ThreadJob – Run jobs in parallel

Common Mistakes

- ✖ Forgetting to install module.
- ✖ Importing module without checking version.
- ✖ Not using -Scope properly.
- ✖ Installing from untrusted sources.
- ✖ Not updating modules regularly.

Module Workflow

1. Find Module – Search online → 2. Install Module – Install from PSGallery → 3. Import Module – Load into session → 4. Use Commands – Run module commands → 5. Remove Module – Unload when done → 6. Update / Manage – Keep modules updated

Example: Using a Module (PSWindowsUpdate)

```
# Install module
Install-Module -Name PSWindowsUpdate -Scope CurrentUser -Force
# Import module
Import-Module PSWindowsUpdate
# Get available updates
Get-WindowsUpdate
# Install updates
Install-WindowsUpdate -AcceptAll -AutoReboot
```

Check Module Information

```
Get-Module -Name PSWindowsUpdate -ListAvailable |
Select-Object Name, Version, Author, Description
```

Output (Sample)

Name	Version	Author	Description
PSWindowsUpdate	2.2.1.3	Manuel Hoefs (jph)	A PowerShell Module to manage Windows Updates.

Summary

PowerShell Modules boost productivity and save time.
Find, Install, Import, and manage modules easily.
Use modules to automate tasks and extend PowerShell power!

Use Modules. Save Time. Automate More. 🚀 | Get-Module • Import-Module • Remove-Module • Find-Module • Install-Module | PowerShell Modules – Do More with Less Code! 🚀

POWERSHELL GALLERY

Manage, Discover & Share PowerShell Modules! 🚀

Find-Module • Install-Module • Update-Module • Publish-Module
PowerShell Gallery • Packages • Modules • Repository

Find-Module	Install-Module	Update-Module	Publish-Module	Packages	Modules	Repository	PowerShell Gallery
Search Modules	Install Package	Update Existing	Publish to Gallery	NuGet Packages	Reusable Code	Module Repository	Online Gallery

What is PowerShell Gallery?

- ✓ Official repository for PowerShell modules
- ✓ Host your own private modules
- ✓ Discover, install, update & publish modules
- ✓ Share your scripts with the community
- ✓ Free & open for everyone!

Syntax

```
# Search for modules
Find-Module <Keyword>

# Install a module
Install-Module <ModuleName>

# Update a module
Update-Module <ModuleName>

# Publish a module
Publish-Module -Path <Path>
  -NuGetApiKey <APIKey>
```

Key Points

- PowerShell Gallery is the official module repository.
- Use Find-Module to search for modules.
- Install modules with Install-Module.
- Keep modules updated with Update-Module.
- Publish your own modules to share.
- Use a NuGet API Key to publish packages.

Common Repositories

Repository	Description
PSGallery	Official PowerShell Gallery
PSGalleryPreview	Preview modules
InternalRepo	Private internal repository
LocalRepo	Local machine repository
CustomRepo	Custom module repository

Pro Tip

- Always check module details before installing.
- Use -Scope CurrentUser for user-level install.
- Update modules regularly for security & bugs.
- Publish modules with proper versioning.
- Automate module management in scripts.

1 Find-Module

Search for modules in the PowerShell Gallery.

```
Find-Module -Name ActiveDirectory
```

Output (Sample)

Version	Name	Repository	Description
2.0.5.0	ActiveDirectory	PSGallery	Active Directory module
1.0.0.47	ActiveDirectoryDsc	PSGallery	DSC resources for AD
0.4.0.3	PSAds	PSGallery	Active Directory cmdlets

Use when:

- You want to find modules by name or keyword.
- You want to explore available modules.

2 Install-Module

Install a module from the PowerShell Gallery.

```
Install-Module -Name ActiveDirectory
```

Output (Sample)

```
Untrusted repository
You are installing the modules from an
untrusted repository. Are you sure you
want to install the modules?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
```

Use when:

- You want to install a module for the current user or system.
- You want to use community modules.

3 Update-Module

Update a module to the latest version.

```
Update-Module -Name ActiveDirectory
```

Output (Sample)

Version	Name	Repository	Status
2.0.5.0	ActiveDirectory	PSGallery	Updated

Use when:

- You want to update installed modules.
- You want the latest features & fixes.

4 Publish-Module

Publish your module to PowerShell Gallery.

```
Publish-Module -Path .\MyModule\
-NuGetApiKey (Get-Content APIKey.txt)
```

Output (Sample)

```
Published MyModule 1.0.0 to
https://www.powershellgallery.com
```

Use when:

- You want to share your module with others.
- You want to contribute to the community.

5 Uninstall-Module

Remove a module from your system.

```
Uninstall-Module -Name ActiveDirectory
```

Output (Sample)

```
Module ActiveDirectory
uninstalled successfully.
```

Use when:

- You no longer need a module.
- You want to remove old or unused modules.

6 List Installed Modules

List all modules installed on your system.

```
Get-Module -ListAvailable
```

Output (Sample)

Version	Name	Repository
2.0.5.0	ActiveDirectory	PSGallery
7.4.0.0	Microsoft.PowerShell.Utility	PSGallery
3.1.0.0	PSReadLine	PSGallery

Use when:

- You want to see installed modules.
- You want to check versions.

7 Install with Scope

Install a module for current user or all users.

```
Install-Module -Name PSReadLine
-Scope CurrentUser
```

Output (Sample)

```
Module PSReadLine 2.3.5
installed successfully.
```

Use when:

- You want user-level installation.
- You want to avoid admin rights.

8 Save Module

Save a module locally without installing.

```
Save-Module -Name PSReadLine
-Path C:\Modules
```

Output (Sample)

```
Module PSReadLine 2.3.5
saved to C:\Modules
```

Use when:

- You want to download modules for offline use.
- You want to share modules manually.

9 Register Repository

Add a custom repository.

```
Register-PSRepository -Name MyRepo
-SourceLocation 'https://myrepo.com/ps'
-InstallationPolicy Trusted
```

Output (Sample)

```
Repository 'MyRepo' registered successfully.
```

Use when:

- You want to use private repositories.
- You want to manage custom sources.

10 Set Repository

Set PSGallery as a trusted repository.

```
Set-PSRepository -Name PSGallery
-InstallationPolicy Trusted
```

Output (Sample)

```
Repository 'PSGallery' is now trusted.
```

Use when:

- You want to install modules without prompts.
- You trust the repository source.

Popular Modules

PSReadLine	Command line editing
Az	Microsoft Azure modules
Pester	Testing framework
Posh-Git	Git prompt & tools
PowerShellGet	Module management
ThreadJob	Run jobs in parallel
SqlServer	SQL Server management
ActiveDirectory	Active Directory cmdlets

Common Mistakes

- ✖ Forgetting to trust PSGallery.
- ✖ Installing modules for the wrong scope.
- ✖ Not updating modules regularly.
- ✖ Ignoring module dependencies.
- ✖ Publishing without proper versioning.

Search Tips with Find-Module

Example	Description
Find-Module -Name Az	Search by exact name
Find-Module -Keyword 'Azure'	Search by keyword
Find-Module -MinimumVersion 2.0	Filter by minimum version
Find-Module -AllVersions	Show all versions
Find-Module -Repository PSGallery	Search in specific repository

Module Details

```
Get-Module -Name PSReadLine -ListAvailable |
Select-Object Name,Version,Author,Description
```

Output (Sample)

Name	Version	Author	Description
PSReadLine	2.3.5	Microsoft Corp.	Command line editing for PowerShell

Dependencies

```
Find-Module -Name Az |
Select-Object -ExpandProperty Dependencies
```

Output (Sample)

Name
Azure.Core
Microsoft.Rest.ClientRuntime

Using Modules in Scripts

```
Import-Module PSReadLine

# Now use module commands
Get-PSReadLineOption
```

Update All Modules

```
Get-Module -ListAvailable |
Where-Object {$_.Version -lt $_.Version} |
Update-Module
```

Use when:

- You want to update all outdated modules.

SewerShell Gallery Workflow

1. Find Module — Search online → 2. Install Module — Install on system → 3. Update Module — Keep up to date → 4. Use Module — Run commands → 5. Publish Module — Share with others

Get Your API Key

1. Go to: https://www.powershellgallery.com
2. Sign in with your Microsoft account
3. Click your profile → API Keys
4. Generate a new API Key
5. Use it with Publish-Module

Summary

PowerShell Gallery makes it easy to discover, install, update and share modules. Use the right commands to save time, automate tasks and boost productivity!

Discover. Install. Update. Share. 🚀 | Find-Module • Install-Module • Update-Module • Publish-Module | PowerShell Gallery – Your Module Hub!

EXECUTION POLICY in PowerShell

Control which scripts run on your system for better security!

Get-ExecutionPolicy	Set-ExecutionPolicy	Restricted	RemoteSigned	AllSigned	Unrestricted	Scripts	Security
Check Current Policy	Change Policy	No Scripts Allowed	Local Scripts OK Downloaded Signed	All Scripts Must Be Signed	All Scripts Allowed	.ps1 Files	Protect Your System

What is Execution Policy?

Execution Policy helps protect your system by controlling the conditions under which PowerShell loads configuration files and runs scripts. It is not a security boundary, but a safety feature.

Syntax

```
# Get current execution policy
Get-ExecutionPolicy

# Set execution policy
Set-ExecutionPolicy -ExecutionPolicy <PolicyName>
    [-Scope <ScopeName>] [-Force] [-Confirm]
)
# Example
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Key Points

- Execution Policy helps prevent running malicious scripts.
- It does NOT prevent running scripts completely.
- It only affects PowerShell scripts (.ps1 files).
- It has different scopes: Process, CurrentUser, LocalMachine.
- Choose the right policy for balance between security & usability.

Scopes

Scope	Description
Process	Affects only the current PowerShell session.
CurrentUser	Affects the current user.
LocalMachine	Affects all users on the computer.
UserPolicy	Set by Group Policy for current user.
MachinePolicy	Set by Group Policy for all users.

Pro Tip

Use RemoteSigned for most use cases.
It allows your local scripts to run and only requires downloaded scripts to be signed.

Execution Policy Levels

Restricted

No scripts are allowed to run.

Value: Restricted

Example

```
Set-ExecutionPolicy
  Restricted
```

What it does:
Blocks all scripts from running.

Use when:
Maximum security is required.

RemoteSigned

Local scripts run without signature. Downloaded scripts must be signed.

Value: RemoteSigned

Example

```
Set-ExecutionPolicy
  RemoteSigned
```

What it does:
Allows local scripts. Downloaded scripts must have a valid signature.

Use when:
Recommended for most users.

AllSigned

All scripts must be signed by a trusted publisher.

Value: AllSigned

Example

```
Set-ExecutionPolicy
  AllSigned
```

What it does:
Requires a valid signature for all scripts.

Use when:
High security environments.

Unrestricted

All scripts can run. No signature required.

Value: Unrestricted

Example

```
Set-ExecutionPolicy
  Unrestricted
```

What it does:
No restrictions on running scripts.

Use when:
You trust all scripts (Not recommended).

Bypass

Nothing is blocked and no warnings.

Value: Bypass

Example

```
Set-ExecutionPolicy
  Bypass
```

What it does:
Skips all warnings and prompts.

Use when:
For automation or CI/CD revionmeents only.

How to Check & Set

Check Current Policy

```
Get-ExecutionPolicy
```

Set Policy (Current User)

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Set Policy (Local Machine)

```
Set-ExecutionPolicy RemoteSigned -Scope LocalMachine
```

Set Policy with Force

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Force
```

Temporary for Current Session

```
Set-ExecutionPolicy Bypass -Scope Process
```

Examples

1. Check Current Policy

```
Get-ExecutionPolicy
```

Output (Example)

```
RemoteSigned
```

2. Set Policy for Current User

```
Set-ExecutionPolicy RemoteSigned
-Scope CurrentUser
```

Output (Example)

```
Execution Policy Change
The execution policy helps protect you
from scripts that yoo do not trust. ...
[Y] Yes [A] Yes to All [N] No [S] Suspend
[?] Help (default is "N"): Y
```

3. Verify the Change

```
Get-ExecutionPolicy -Scope
CurrentUser
```

Output (Example)

```
RemoteSigned
```

4. Set Policy with Force

```
Set-ExecutionPolicy RemoteSigned
-Scope CurrentUser -Force
```

Output (Example)

```
(No output if successful)
```

Quick Reference

Command	Description
Get-ExecutionPolicy	Get current execution policy
Set-ExecutionPolicy	Set execution policy
-Scope	Define the scope (Process, CurrentUser, LocalMachine)
-Force	Set policy without confirmation
-Confirm	Ask for confirmation
-WhatIf	Show what would happen

Common Mistakes

- Using Unrestricted in production systems.
- Forgetting -Scope leads to unexpected changes.
- Not understanding the difference between local and downloaded scripts.
- Ignoring warnings without understanding risks.
- Disabling policy completely (Bypass) without reason.

Security First!
Use the most restrictive policy that allows you to get your work done.

Recommended Policy for Most Users
RemoteSigned
Allows productivity while keeping you safe from untrusted scripts.

Need Help?
Run Get-Help about_Execution_Policies for detailed documentation.

Remember:
Execution Policy is not a security boundary, but a safety feature to protect from accidental script execution.

Secure. Control. Automate. Get-ExecutionPolicy • Set-ExecutionPolicy • Restricted • RemoteSigned • AllSigned • Unrestricted • Bypass | PowerShell Execution Policy – Stay Safe, Stay in Control!

POWERSHELL AUTOMATION

Automate Tasks. Save Time. Work Smarter!

Files • Processes • Services • Users • Network • Reports • Scheduled Tasks • Automation

Files	Processes	Services	Users	Network	Reports	Scheduled Tasks	Automation
Manage Files & Folders	Control Processes	Manage Services	Manage Local & AD Users	Network Info & Tools	Generate Reports	Schedule & Automate Tasks	Automate Everything

Why Automate with PowerShell?

- Save time by automating repetitive tasks
- Reduce human errors
- Manage systems efficiently
- Generate powerful reports
- Improve productivity & consistency
- Do more with less effort!

Syntax

```
# Automation Example
Get-Service | Where-Object {$_.Status -eq 'Running'}

Get-Process | Sort-Object CPU -Descending |
Select-Object -First 5

Get-ChildItem -Path "C:\Logs" -Recurse -File |
  Sort-Object LastWriteTime -Descending

Get-ScheduledTask | Where-Object {$_.State -eq 'Ready'}
```

Automation Benefits

- Consistent & repeatable tasks
- Quick system administration
- Better reporting & monitoring
- Easy scheduling with Task Scheduler
- Works on local & remote systems

Pro Tip

Use scripts with Task Scheduler for fully automated workflows. Combine with email reports for complete automation!

Common Cmdlets for Automation

Category	Useful Cmdlets
Files	Get-ChildItem, Copy-Item, Move-Item, Remove-Item
Processes	Get-Process, Start-Process, Stop-Process
Services	Get-Service, Start-Service, Stop-Service, Set-Service
Users	Get-LocalUser, New-LocalUser, Add-LocalGroupMember
Network	Get-NetIPConfiguration, Test-Connection, Get-NetTCPConnection
Reports	Export-Csv, ConvertTo-Html, Out-File, Out-Printer
Scheduled Tasks	Get-ScheduledTask, Register-ScheduledTask, Start-ScheduledTask
Automation	Invoke-Command, Send-MailMessage, Start-Job

Files

Manage Files & Folders

```
# List files
Get-ChildItem

# Find files
Get-ChildItem -Filter *.log -Recurse

# Copy files
Copy-Item "C:\Source" "C:\Backup"
-Recurse

# Delete old files (30+ days)
Get-ChildItem -File | Where-Object
($_.LastWriteTime -lt (Get-
Date).AddDays(-30)) |
Remove-Item -Force
```

Use for:

- File management
- Log cleanup
- Backup automation
- Report generation

Processes

Manage & Monitor Processes

```
# List all processes
Get-Process

# Top 5 by CPU
Get-Process | Sort-Object CPU -Desc |
Select-Object -First 5

# Stop a process
Stop-Process -Name "notepad"

# Start a process
Start-Process "chrome"
```

Use for:

- Process monitoring
- Kill unresponsive apps
- Resource tracking
- Performance reports

Services

Manage Windows Services

```
# List services
Get-Service

# Start a service
Start-Service -Name "Spooler"

# Stop a service
Stop-Service -Name "wuauserv"

# Set startup type
Set-Service -Name "wuauserv"
-StartupType Automatic
```

Use for:

- Service management
- Health monitoring
- Startup configuration
- Automated recovery

Users

Manage Users & Accounts

```
# List local users
Get-LocalUser

# Create a user
New-LocalUser -Name "John"
-Password (ConvertTo-SecureString
"Pass@123" -AsPlainText -Force)

# Add to group
Add-LocalGroupMember -Group
"Users" -Member "John"

# Disable user
Disable-LocalUser -Name "Guest"
```

Use for:

- User management
- Account reports
- Bulk user operations
- Security automation

Network

Network Tools & Info

```
# IP configuration
Get-NetIPConfiguration

# Test connectivity
Test-Connection google.com

# Open ports
Get-NetTCPConnection | Where-
Object {$_.State -eq "Listen"}

# Network adapters
Get-NetAdapter
```

Use for:

- Network monitoring
- Connectivity checks
- Port scanning
- Adapter management

Reports

Generate & Export Reports

```
# System info report
Get-ComputerInfo | Export-Csv
"C:\Reports\SystemInfo.csv" -NoTypeInformation

# Disk space report
Get-PSDrive -PSProvider FileSystem |
Select-Object Name,@{N='Free(GB)';
E={[math]::Round($_.Free/1GB,2)}},
@{N=[Used]::Round(($_.Used)/1GB,2)}} |
Export-Csv "C:\Reports\DiskReport.csv"
-NoTypeInformation

# Installed programs report
NetworkProperty HKLM:\Software\Microsoft\
Windows\Windows\CurrentVersion\Uninstall\* |
Select-Object DisplayName, DisplayVersion |
Export-Csv "C:\Reports\Programs.csv"
-NoTypeInformation
```

Use for:

- System reports
- Compliance reports
- Inventory reports
- Scheduled reporting

Scheduled Tasks

Schedule & Automate Tasks

```
# List scheduled tasks
Get-ScheduledTask

# Create a task
$Action = New-ScheduledTaskAction
-Execute "PowerShell.exe" -Argument
"-File C:\Scripts\Cleanup.ps1"
$Trigger = New-ScheduledTaskTrigger
-Daily -At 03:00AM
Register-ScheduledTask -TaskName
-Action $Action -Trigger $Trigger
# Run a task
Start-ScheduledTask -TaskName
"DailyCleanup"
# Remove a task
Unregister-ScheduledTask -TaskName
"DailyCleanup" -Confirm:$false
```

Use for:

- Task scheduling
- Script automation
- Maintenance tasks
- Automated workflows

Automation

Automate Everything

```
# Run multiple tasks
C:\Scripts\Backup.ps1

C:\Scripts\Cleanup.ps1

C:\Scripts\Report.ps1

# Send email report
Send-MailMessage -To "admin@example.com"
-From "reporter@domain.com"
-Subject "Daily Report" -Body "Please find
attached report." -Attachments
"C:\Reports\*.csv"
-SmtpServer "smtp.domain.com"

# Logging
"$(Get-Date) - Automation completed" |
Out-File "C:\Logs\Automation.log"
```

Use for:

- End-to-end automation
- Email notifications
- Logging & auditing
- Workflow orchestration

Automation Workflow Example

1. Collect Data — Get system info, logs, and status
2. Process Data — Filter, sort, and analyze data
3. Generate Report — Create reports in CSV/HTML format
4. Send Report — Email reports to admin/users
5. Schedule Task — Run automatically at defined time

```
# Example: Daily System Report Automation
Get-ComputerInfo | Export-Csv "C:\Reports\SystemInfo.csv" -NoTypeInformation
Get-PSDrive -PSProvider FileSystem | Export-Csv "C:\Reports\DiskReport.csv" -NoTypeInformation
Send-MailMessage -To "admin@example.com" -From "reporter@domain.com" -Subject "Daily System Report" \
 -Body "Please find the daily system reports." -Attachments "C:\Reports\*.csv" -SmtpServer "smtp.domain.com"
"$(Get-Date) - Report sent successfully" | Out-File "C:\Logs\Automation.log" -Append
```

Best Practices

- Always test scripts before scheduling.
- Use logging for all automated tasks.
- Keep scripts modular and reusable.
- Handle errors using try/catch.
- Use meaningful names for tasks & files.
- Secure sensitive data (passwords, keys).
- Regularly review and update automation.

Common Mistakes

- Running scripts without testing.
- Hardcoding sensitive information.
- Ignoring error handling.
- Not logging activities.
- Over-privileged accounts usage.
- Skipping script documentation.
- Not monitoring scheduled tasks.

Quick Reference

Task	Command
List Files	Get-ChildItem
Top CPU Processes	Get-Process | Sort-Object CPU -Desc
List Services	Get-Service
IP Configuration	Get-NetIPConfiguration
System Report	Get-ComputerInfo
Scheduled Tasks	Get-ScheduledTask

Automate Today. Save Tomorrow! | Files | Processes | Services | Users | Network | Reports | Scheduled Tasks | Automation | PowerShell Automation – Do More, Effort Less!

POWERSHELL QUICK REFERENCE

Essential Commands, Scripts & Automation at a Glance! ⚡

Get-Help	Get-Command	Get-Process	Get-Service	Get-ChildItem	Where-Object	ForEach-Object	Invoke-Command	Scripts	Automation
Help System	All Commands	Running Processes	Windows Services	Files & Folders	Filter Objects	Process Each Item	Run Remote Cmds	.ps1 Files	Automate Tasks

1 Get-Help

Get help about cmdlets, functions, scripts and PowerShell concepts.

Syntax

```
Get-Help <Name>
Get-Help <Name> -Detailed
Get-Help <Name> -Examples
```

Example

```
Get-Help Get-Process -Examples
Get-Help about_Automation
```

2 Get-Command

Get all commands. Find specific cmdlets, functions, aliases and scripts.

Syntax

```
Get-Command
Get-Command <Name>
Get-Command -Module <ModuleName>
```

Example

```
Get-Command
Get-Command Get-Service
Get-Command -Module Microsoft.PowerShell.Management
```

3 Get-Process

Get the list of running processes on the local or remote computer.

Syntax

```
Get-Process
Get-Process <Name>
Get-Process | Where-Object ...
```

Example

```
Get-Process
Get-Process chrome
Get-Process | Where-Object {$_.CPU -gt 100}
```

4 Get-Service

Get the list of services and their status on the system.

Syntax

```
Get-Service
Get-Service <Name>
Get-Service | Where-Object ...
```

Example

```
Get-Service
Get-Service Spooler
Get-Service | Where-Object {$_.Status -eq 'Running'}
```

5 Get-ChildItem

Get files and folders. Works like Dir or ls in other shells.

Syntax

```
Get-ChildItem [Path]
Get-ChildItem -Recurse
Get-ChildItem -Filter <Filter>
```

Example

```
Get-ChildItem  C:\Temp
Get-ChildItem  C:\ -Recurse  -File
Get-ChildItem  C:\Logs -Filter  *.log
```

⚡ Common Shortcuts

↑	Previous Command
↓	Next Command
Tab	Auto-complete
Ctrl + C	Cancel
Ctrl + L	Clear Screen
Ctrl + R	Search History
F7	Show Command History
Ctrl + D	Exit PowerShell

Where-Object

Filter objects based on a condition.

Syntax

```
<Command> | Where-Object {  <Condition> }
<Command> | Where-Object Property -eq Value
```

Example

```
Get-Process | Where-Object {$_.CPU -gt 100}
Get-Service | Where-Object Status -eq 'Running'
```

7 ForEach-Object

Perform an operation on each object in the pipeline.

Syntax

```
<Command> | ForEach-Object {  <Action> }
<Command> | ForEach-Object -Process { <Action> }
```

Example

```
Get-ChildItem *.txt | ForEach-Object { $_.Name }
Get-Process | ForEach-Object { $_.Kill() }
```

Invoke-Command

Run commands on remote computers.

Syntax

```
Invoke-Command -ComputerName <Name> -ScriptBlock { <Command> }
Invoke-Command -ComputerName <Name> -FilePath <ScriptPath>
```

Example

```
Invoke-Command -ComputerName PC01 -ScriptBlock { Get-Process }
Invoke-Command -ComputerName PC01 -FilePath C:\Scripts\Check.ps1
```

9 Scripts (.ps1)

Create and run PowerShell scripts to automate tasks.

Syntax

```
.\ScriptName.ps1
& .\ScriptName.ps1 -Param value
PowerShell -File .\ScriptName.ps1
```

Example

```
.\Backup.ps1
.\Backup.ps1 -Path C:\Data
PowerShell -File .\Report.ps1
```

10 Automation

Automate repetitive tasks and improve productivity.

Syntax

```
Schedule tasks with Task Scheduler
Use scripts, functions & modules
Use loops, conditions & pipelines
```

Example

```
Schedule daily backup using script
Email report automatically
Monitor system & send alerts
```

Pipeline Examples

```
Get-Process | Where-Object {$_.CPU -gt 100}
Get-Service | Where-Object Status -eq 'Running'
Get-ChildItem C:\ | Where-Object {$_.Length -gt 1MB}
Get-Process | Sort-Object CPU -Descending
Get-Service | Select-Object Name, Status, StartType
```

$ Useful Variables

$env:USERNAME	Current user name
$env:COMPUTERNAME	Computer name
$env:TEMP	Temp folder path
$env:PATH	System PATH
$PSVersionTable	PS version info

Sample Automation Script

Daily Report Script Example

```
# Daily System Report
$reportPath = "C:\Reports\DailyReport.txt"
$time = Get-Date
"Daily Report - $time" | Out-File $reportPath
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10 |
Out-File $reportPath -Append
Get-Service | Where-Object Status -eq 'Running' |
Select-Object Name, Status, StartType | Out-File $reportPath -Append
"Report generated at $reportPath"
```

Automation Workflow

1. **Plan** — Identify tasks to automate →
2. **Script** — Write PowerShell script →
3. **Schedule** — Use Task Scheduler or triggers →
4. **Execute** — Run script automatically →
5. **Report/Notify** — Generate reports & send alerts

💡 Automate Today, Save Time Tomorrow!

Quick Examples

```
Get-Process notepad
Get-Service | Sort-Object Status
Get-ChildItem C:\Windows -Recurse -File | Select-Object FullName, Length
Get-Process | Where-Object {$_.CPU -gt 50} | Stop-Process -Force
Invoke-Command -ComputerName PC01 -ScriptBlock { Get-Process }
Get-Service | Export-Csv C:\Reports\Services.csv -NoTypeInformation
```

★ Pro Tips

- ✅ Use Tab for quick auto-complete.
- ✅ Use Get-Help for any command help.
- ✅ Pipelines (|) are very powerful.
- ✅ Write scripts to save your time.
- ✅ Test scripts before automating.

PowerShell: Powerful. Flexible. Automated. | Get-Help | Get-Command | Get-Process | Get-Service | Get-ChildItem | Where-Object | ForEach-Object | Invoke-Command | Scripts | Automation |